7.1 Sequences

This section has been set as optional by your instructor.

A ***sequence*** is a special type of function in which the domain is a consecutive set of integers. For example, a sequence can be defined to denote a student's GPA for each of the four years the student attended college. The domain of the function is {1, 2, 3, 4} for each of the four years. The function g is specified by the numerical values for the GPA in each year:

$$g(1) = 3.67, \quad g(2) = 2.88, \quad g(3) = 3.25, \quad g(4) = 3.75$$

When a function is specified as a sequence, using subscripts to denote the input to the function is more common, so g_k is used instead of g(k). The expression g_k is called a ***term*** of a sequence, and the variable k is the ***index*** of g_k. The sequence is written:

$$g_1 = 3.67, \quad g_2 = 2.88, \quad g_3 = 3.25, \quad g_4 = 3.75$$

When the variable g and indices are understood (or not important), the sequence can be specified by listing just the terms: 3.67, 2.88, 3.25, 3.75.

The entire sequence is denoted by $\{g_k\}$, whereas g_k (without the curly braces) is used to denote a single term in the sequence. While the first index is commonly 0 or 1, a sequence may start with any integer, as in the following example:

$$a_{-2} = 0, \quad a_{-1} = 1, \quad a_0 = 1, \quad a_1 = 0$$

A sequence with a finite domain is called a ***finite sequence***. In a finite sequence, there is an ***initial index*** m and a ***final index*** n, where $n \geq m$. Then a_m is the ***initial term*** and a_n is the ***final term***.

$$a_m, \quad a_{m+1}, \quad \ldots \quad , a_n$$

A sequence with an infinite domain is called an ***infinite sequence***. In an infinite sequence, the indices go to infinity in the positive direction. There is an initial index m, and the sequence is defined for all indices k such that $k \geq m$:

$$a_m, \quad a_{m+1}, \quad a_{m+2}, \quad \ldots$$

A sequence can be specified by an ***explicit formula*** showing how the value of term a_k depends on k. For example, $d_k = 2^k$ for $k \geq 1$. The infinite sequence $\{d_k\}$ starts with: 2, 4, 8, 16, ...

PARTICIPATION ACTIVITY 7.1.1: Sequence basics.

Animation captions:

1. Finite sequence example: $S_1 = 2, S_2 = 5, S_3 = -1, S_4 = 7, S_5 = 8$. The sequence $\{S_k\}$ is finite. The indices are 1, 2, 3, 4, 5.
2. The terms are 2, 5, -1, 7, 8.
3. 1 is the initial index. The initial term is 2. 5 is the final index. The final term is 8.
4. $\{a_k\}$ is an infinite sequence defined by $a_k = \frac{1}{k}$, for k = 1, 2, 3, ... $\{a_k\}$ starts with 1, 1/2, 1/3, ... a_1 = 1 is the initial term in the sequence.

PARTICIPATION ACTIVITY 7.1.2: Sequence basics.

Define the sequence $\{a_k\}$, beginning with $a_1 = 7$:

$$7, \quad 10, \quad 13, \quad 16, \quad 19, \quad 22, \quad 25$$

1) What is a_2?

Check Show answer

2) What is the index of the term 16 in the sequence?

Check Show answer

3) What is the final index of the sequence?

Check Show answer

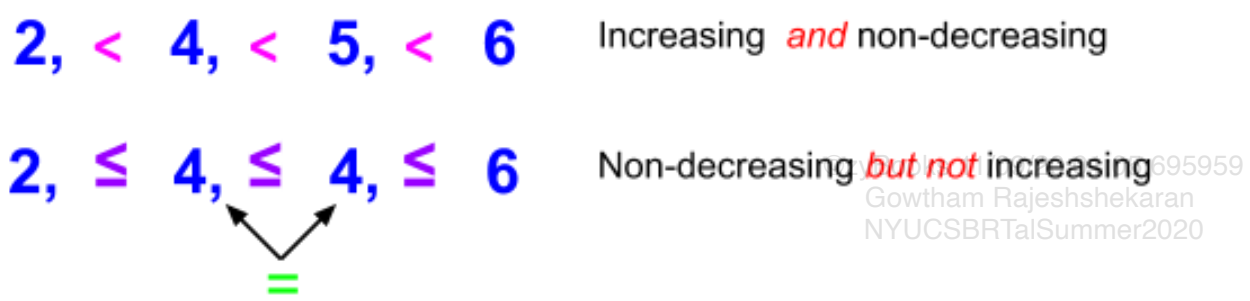
4) Consider the sequence defined by the formula $b_k = k^2$ for $k \geq 2$. What is the third term in the sequence?

Check Show answer

Increasing and decreasing sequences

A sequence is ***increasing*** if for every two consecutive indices, k and k + 1, in the domain, $a_k < a_{k+1}$. A sequence is ***non-decreasing*** if for every two consecutive indices, k and k + 1, in the domain, $a_k \leq a_{k+1}$. Notice that an increasing sequences is always non-decreasing as well, because if $a_k < a_{k+1}$ then it is also true that $a_k \leq a_{k+1}$.

Figure 7.1.1: Increasing and non-decreasing sequences.



A sequence is ***decreasing*** if for every two consecutive indices, k and k + 1, in the domain, $a_k > a_{k+1}$. A sequence is ***non-increasing*** if for every two consecutive indices, k and k + 1, in the domain, $a_k \geq a_{k+1}$.

Figure 7.1.2: Decreasing and non-increasing sequences.

PARTICIPATION ACTIVITY 7.1.3: Increasing and decreasing sequences.

Select the choice that describes the sequence.

1) 1, 2, 3, 3

- Non-decreasing
- Non-increasing
- Increasing

2) 4, 3.5, 2, -1

- Non-increasing but not decreasing
- Both non-increasing and non-decreasing
- Decreasing and non-increasing

3) 2, 2, 2, 2

- Neither non-increasing nor non-decreasing
- Non-increasing and non-decreasing
- Non-increasing but not non-decreasing

Geometric sequences and arithmetic sequences are two types of sequences that arise frequently in discrete mathematics. Both types of sequences are described below.

Geometric sequences

Definition 7.1.1: Definition of a geometric sequence.

A ***geometric sequence*** is a sequence of real numbers where each term after the initial term is found by taking the previous term and *multiplying* by a fixed number called the ***common ratio***. A geometric sequence can be finite or infinite.

PARTICIPATION ACTIVITY 7.1.4: Geometric sequences.

Animation captions:

1. Geometric sequence example: the initial term = 4, and the common ratio = 1/2. The second term is the initial term times the common ratio: $4 \times \frac{1}{2} = 2$.
2. The next term is $2 \times \frac{1}{2} = 1$ and the one after that is $1 \times \frac{1}{2} = \frac{1}{2}$, etc

The infinite geometric sequence with initial term $a_0 = 1$ and common ratio r = 1/2 begins with:

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$$

The finite geometric sequence with initial term $a_0 = 5$, final index 6, and common ratio r = -1 is:

$$5, -5, 5, -5, 5, -5, 5$$

If the initial index of an infinite geometric sequence $\{s_k\}$ is 0, then the explicit formula defining the sequence is: $s_k = a \cdot r^k$, for $k \geq 0$.

Example 7.1.1: Interest rates and geometric sequences.

Money in a bank account earning a fixed rate of interest can be expressed as a geometric sequence. Suppose $1000 is stored in a bank account that earns 6% annual interest compounded monthly. Since the interest rate is annual and compounded monthly, (6/12)% of the current amount is added to the account each month. a_0=1000 is the initial balance in the account, and a_n is the balance in the account after n months of earning interest. Each month, the balance in the account is 1.005 times the amount that was in the account in the previous month. The sequence $\{a_n\}$ is a geometric sequence with $a_n = 1000 \cdot (1.005)^n$ for $n \geq 0$.

PARTICIPATION ACTIVITY 7.1.5: Geometric sequences.

1) Let $\{s_n\}$ be a geometric sequence that starts with an initial index of 0. The initial term is 2 and the common ratio is 5. What is s_2?

Check Show answer

2) Let $\{s_n\}$ be a geometric sequence that starts with an initial index of 0. The initial term is 16 and the common ratio is 1/2. What is s_3?

Check Show answer

3) Consider the geometric sequence: 3, 6, 12,
What is the common ratio?

Check Show answer

Arithmetic sequences

Definition 7.1.2: Definition of an arithmetic sequence.

An ***arithmetic sequence*** is a sequence of real numbers where each term after the initial term is found by taking the previous term and *adding* a fixed number called the ***common***

difference. An arithmetic sequence can be finite or infinite.

PARTICIPATION ACTIVITY 7.1.6: Arithmetic sequences.

Animation captions:

1. Arithmetic sequence example: the initial value = -1, and the common difference = 2.5. The second term is: -1 + 2.5 = 1.5.
2. The next term is 1.5 + 2.5 = 4 and the one after that is 4 + 2.5 = 6.5, etc.

The infinite arithmetic sequence with initial term $a_0 = 2$ and common difference d = 3 begins with:

$$2, \quad 5, \quad 8, \quad 11, \quad \ldots$$

The finite arithmetic sequence with initial term $a_0 = 3$, final index 5, and common difference d = -2 is:

$$3, \quad 1, \quad -1, \quad -3, \quad -5, \quad -7$$

If the initial index of an arithmetic sequence $\{t_n\}$ is 0, then the explicit formula defining the sequence is: $t_n = t_0 + dn$, for $n \geq 0$, where d is the common difference .

Example 7.1.2: Accumulated resources and arithmetic sequences.

Suppose a person inherits a collection of 500 baseball cards and decides to continue growing the collection at a rate of 10 additional cards each week. a_n is the number of cards in the collection after n weeks of collecting. Since the collection starts with 500 cards, $a_0 = 500$. The sequence $\{a_n\}$ is an arithmetic sequence with an initial value of 500 and a common difference of 10. After n weeks of collecting, $a_n = 500 + 10n$.

PARTICIPATION ACTIVITY 7.1.7: Arithmetic sequences.

1) Let $\{s_n\}$ be an arithmetic sequence that starts with an initial index of 0. The initial term is 3 and the common difference is -2. What is s_2?

Check Show answer

2) Consider the arithmetic sequence: 7, 4, 1, ... What is the next term in the

sequence?

Check Show answer

CHALLENGE ACTIVITY 7.1.1: Sequences.

Additional exercises

Exercise 7.1.1: Evaluating sequences.

About

Give the first ten terms of the following sequences. You can assume that the sequences start with an index of 1. Logs are to base 2.

Indicate whether the sequence is increasing, decreasing, non-increasing, or non-decreasing. The sequence may have more than one of those properties.

(a) The n^{th} term is $\lceil \sqrt{n} \rceil$.

(b) The first two terms in the sequence are 1. The rest of the terms are the sum of the two preceding terms.

(c) The n^{th} term is the largest integer k such that $k! \leq n$.

(d) The n^{th} term is $1/n$.

(e) The n^{th} term is 3.

(f) The n^{th} term is n^2.

(g) The n^{th} term is $\lceil \log n \rceil$.

(h) The n^{th} term is $2^{\lceil \log n \rceil}$.

(i) The n^{th} term is $\lceil \frac{-n}{2} \rceil$.

Exercise 7.1.2: Arithmetic and geometric sequences.

About

Give the first six terms of the following sequences. You can assume that the sequences start with an index of 1.

(a) A geometric sequence in which the first value is 2 and the common ratio is 3.

(b) An arithmetic sequence in which the first value is 2 and the common difference is 3.

(c) A geometric sequence in which the first value is 27 and the common ratio is 1/3.

(d) An arithmetic sequence in which the first value is 3 and the common difference is -1/2.

7.2 Recurrence relations

This section has been set as optional by your instructor.

Some sequences are most naturally defined by specifying one or more initial terms and then giving a rule for determining subsequent terms from earlier terms in the sequence. A rule that defines a term a_n as a function of previous terms in the sequence is called a ***recurrence relation***. For example, in an arithmetic sequence, each number is obtained by adding a fixed value to the previous number.

Example 7.2.1: Recurrence relation defining an arithmetic sequence.

$a_0 = a$ (initial value)
$a_n = d + a_{n-1}$ for $n \geq 1$ (recurrence relation)

Initial value = a. Common difference = d.

Geometric sequences are also naturally specified by recurrence relations. In a geometric sequence, each number is obtained by multiplying the previous number by a fixed constant.

Example 7.2.2: Recurrence relation defining an geometric sequence.

$a_0 = a$ (initial value)
$a_n = r \cdot a_{n-1}$ for $n \geq 1$ (recurrence relation)

Initial value = a. Common ratio = r.

Leonardo Fibonacci, a prominent mathematician in the middle ages, used a recurrence relation to define a sequence that is now known as the ***Fibonacci sequence***. Fibonacci defined the sequence in an attempt to mathematically describe the population growth of rabbits. The colony starts with one pair of newborn rabbits. The rabbits must be at least one month old before they can reproduce. Every pair of reproducing rabbits gives birth to a new pair of rabbits, one male and one female over the course of a month.

Suppose that f_n is the number of pairs of rabbits at the end of month n. Assume that the first pair of rabbits obtained for the colony are born at the end of month 1, so $f_0 = 0$ and $f_1 = 1$. In month n, the colony has all the rabbits from the previous month (f_{n-1}) plus all the new rabbits born in that month. Since each pair of reproducing rabbits gives rise to a new pair, the number of new rabbits appearing in month n is the same as the number of pairs of rabbits that are able to reproduce in month n, which is f_{n-2}, the number of pairs that were alive at least one full month before the beginning of month n. Thus, the Fibonacci sequence is defined as:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2$$

Interestingly, the Fibonacci sequence also arises naturally in computer science in analyzing certain data structures and algorithms for data search. Recurrence relations are used to describe phenomena in many diverse areas of business, engineering, and science. Recurrence relations are especially useful in computer science for modeling the running time of recursive algorithms.

PARTICIPATION ACTIVITY 7.2.1: The Fibonacci sequence.

Animation captions:

1. The initial values for the Fibonacci sequence are $f_0 = 0$ and $f_1 = 1$.
2. $f_2 = f_1 + f_0 = 1 + 0 = 1$
3. $f_3 = f_2 + f_1 = 1 + 1 = 2$
4. $f_4 = f_3 + f_2 = 2 + 1 = 3$

PARTICIPATION ACTIVITY 7.2.2: Recurrence relations.

Define the sequence $\{b_n\}$ as:

$$b_0 = 1$$
$$b_1 = 1$$
$$b_n = b_{n-1} \cdot b_{n-2} + 2 \quad \text{for } n \geq 2$$

1) What is b_2?

Check Show answer

2) What is b_3?

Check Show answer

CHALLENGE ACTIVITY 7.2.1: Recurrence Relations.

Note: You may want to use paper and pencil, and/or a calculator.

The number of initial values required to define a sequence using a recurrence relation depends on which previous terms in the sequence are used to define the n^{th} term. In the Fibonacci sequence, f_n depends on the previous two terms, so the definition of the Fibonacci sequence includes the two initial values for the sequence. f_0 and f_1 are both required to define f_2. If only f_0 was given, then the recurrence relation could not be applied to $f_1 = f_0 + f_{-1}$ because f_{-1} is not defined.

PARTICIPATION ACTIVITY 7.2.3: Initial values for recurrence relations.

Each question below gives a recurrence relation. How many initial values must be given for the sequence to be well defined for all $n \geq 0$?

1) $c_n = 3 \cdot c_{n-1} + 2 \cdot c_{n-2}$

Check Show answer

2) $d_n = (n-1) \cdot d_{n-2}$

Check Show answer

3) $b_n = b_{n-3} \cdot b_{n-2}$

Check Show answer

Fibonacci's rabbit colony is an example of a dynamical system. A ***dynamical system*** is a system that changes over time. Moreover, the state of the system at any point in time is determined by a set of well defined rules that depend on the past states of the system. In a ***discrete time dynamical system***, time is divided into discrete time intervals and the state of the system stays fixed within each time interval. The state during one time interval is a function of the state in previous time intervals. Thus, the history of the system is defined by a sequence of states, indexed by the non-negative integers.

In the case of Fibonacci's rabbits, each time interval is a month and the state of the system is simply the number of pairs of rabbits. Dynamical systems are useful in studying phenomena in biology, engineering, physics, economics, and finance. Many dynamical systems naturally give rise to recurrence relations that describe how the system changes over time.

Example 7.2.3: Outstanding balance on a car loan.

An individual takes out a $20,000 car loan. The interest rate for the loan is 3%, compounded monthly. He wishes to make a monthly payment of $500. Define a_n to be the amount of outstanding debt after n months. Since the interest rate describes the annual interest, the percentage increase each month is actually 3% / 12 = 0.25%. Thus, the multiplicative factor increase each month is 1.0025. The recurrence relation for $\{a_n\}$ is:

$$a_0 = \$20,000$$
$$a_n = (1.0025) \cdot a_{n-1} - 500 \quad \text{for } n \geq 1$$

The $(1.0025) \cdot a_{n-1}$ term describes the increase in debt based on the interest rate. The -500 term describes the decrease due to the monthly payment. The first few values, to the nearest dollar, for the sequence $\{a_n\}$ are:

$$a_0 = \$20,000$$
$$a_1 = \$19,550$$
$$a_2 = \$19,099$$
$$a_3 = \$18,647$$
$$\ldots$$

PARTICIPATION ACTIVITY 7.2.4: Defining sequences by recurrence relations.

Each question describes a dynamical system and the corresponding sequence. Select the recurrence relation that describes the sequence.

1) Let s_n be the salary of an employee in year n. The employee receives a $1000 bonus in December. The salary for the next year is 5% more than the income received by the employee in the previous year (including salary and bonus).
   - $s_n = 1.05 \cdot (s_{n-1}) + 1000$
   - $s_n = 1.5 \cdot (s_{n-1} + 1000)$
   - $s_n = 1.05 \cdot (s_{n-1} + 1000)$

2) 25% of the yeast cells in a petri dish divide every hour. When a single cell divides, it becomes two cells instead of one and the number of cells goes up by one. Let y_n be the number of yeast cells in the petri dish after n hours.
   - $y_n = (0.25)\ y_{n-1}$
   - $y_n = (1.25)\ y_{n-1}$
   - $y_n = (.25)\ y_{n-1} + y_{n-2}$

Additional exercises

Exercise 7.2.1: Evaluating recursively defined sequences.

About

Give the first six terms of the following sequences.

(a) The first term is 1 and the second term is 2. The rest of the terms are the product of the two preceding terms.

(b)
$a_1 = 1$, $a_2 = 5$, and $a_n = 2 \cdot a_{n-1} + 3 \cdot a_{n-2}$ for $n \geq 3$.

(c)
$g_1 = 2$ and $g_2 = 1$. The rest of the terms are given by the formula $g_n = n \cdot g_{n-1} + g_{n-2}$.

(d)
$c_1 = 4$, $c_2 = 5$, and $c_n = c_{n-1} \cdot c_{n-2}$ for $n \geq 2$.

(e)
$b_1 = 1$, $b_2 = 3$, and $b_n = b_{n-1} - 7 \cdot b_{n-2}$ for $n \geq 2$.

(f)
$d_1 = 1$, $d_2 = 1$, and $d_n = (d_{n-1})^2 + d_{n-2}$ for $n \geq 2$.

(g)
$f_1 = 0$, $f_2 = 2$, and $f_n = 5 \cdot f_{n-1} - 2 \cdot f_{n-2}$ for $n \geq 2$.

Exercise 7.2.2: Refinancing a loan.

About

Suppose someone takes out a home improvement loan for $30,000. The annual interest on the loan is 6% and is compounded monthly. The monthly payment is $600. Let a_n denote the amount owed at the end of the nth month. The payments start in the first month and are due the last day of every month.

(a)
Give a recurrence relation for a_n. Don't forget the base case.

(b) Suppose that the borrower would like a lower monthly payment. How large does the monthly payment need to be to ensure that the amount owed decreases every month?

Exercise 7.2.3: A variation on Fibonacci's rabbits.

About

(a)
Give a recurrence relation for the population of Fibonacci's rabbits if the rabbits must be at least two months old to reproduce.
The colony starts with one pair of newborn rabbits. Every pair of reproducing rabbits gives birth to a new pair of rabbits, one male and one female over the course of a month. Let g_n denote the number of pairs of rabbits at the end of month n. Assume

that the first pair of rabbits obtained for the colony are born at the end of month 1.

7.3 Summations

This section has been set as optional by your instructor.

Summation notation is used to express the *sum* of terms in a numerical sequence. Consider a sequence:

$$a_s,\ a_{s+1},\ \ldots\ ,\ a_t$$

The notation to express the sum of the terms in that sequence is:

$$\sum_{i=s}^{t} a_i = a_s + a_{s+1} + \cdots + a_t$$

In the summation $\sum_{i=s}^{t} a_i$, the variable i is called the ***index*** of the summation. The variable s is the ***lower limit*** and the variable t is the ***upper limit*** of the summation. Any variable name could be used for the index instead of i, but variables i, j, k, and l are the most common. The capital letter sigma (Σ) is used to denote the fact that the terms are to be added together. Summation notation can be used to sum up the terms in a sequence defined by an explicit formula:

$$a_n = n^3, \quad \text{for } n = 1, 2, 3, 4$$

Then:

$$\sum_{j=1}^{4} j^3 = 1^3 + 2^3 + 3^3 + 4^3 = 1 + 8 + 27 + 64 = 100$$

The expression $\sum_{j=1}^{4} j^3$ is called the ***summation form*** of the sum, and the expression $1^3 + 2^3 + 3^3 + 4^3$ is called the ***expanded form*** of the sum. When the expression to be summed has more than one term, it is important to use parentheses to indicate that all the terms are included in the summation as in $\sum_{j=1}^{n} (j + 1)$ because the expression $\sum_{j=1}^{n} j + 1$ is interpreted as $\left(\sum_{j=1}^{n} j\right) + 1$.

PARTICIPATION ACTIVITY 7.3.1: Computing summations.

Animation captions:

1. $\sum_{j=-1}^{2} (j^2 + 1) = ((-1)^2 + 1) + (0^2 + 1) + (1^2 + 1) + (2^2 + 1) = 10$

PARTICIPATION ACTIVITY 7.3.2: Computing summations.

Give numerical values for the following summations.

1) $\sum_{j=2}^{5}(2j-1)$

Check Show answer

2) $\sum_{j=0}^{2}(j+1)^2$

Check Show answer

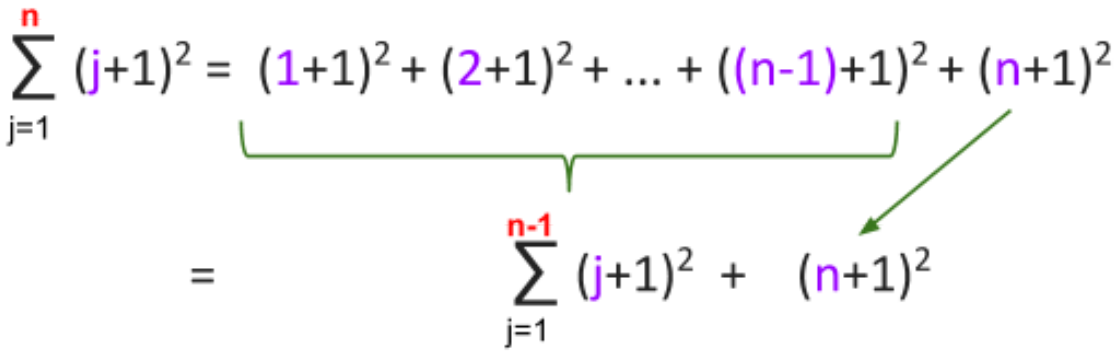
Pulling out a final term from a summations

In working with summations, it is often useful to pull out or add in a final term to a summation:

$$\sum_{k=m}^{n} a_k = \sum_{k=m}^{n-1} a_k + a_n, \quad \text{for} \quad n > m$$

The drawing below illustrates the idea with an example. The expanded form of the sum is given to show why the two expressions are equal:

Figure 7.3.1: Pulling out a final term from a summation.



PARTICIPATION ACTIVITY 7.3.3: Removing a final term in a sum.

Select the choice whose value is equal to the given sum.

1) $\sum_{j=0}^{n}2^j$

- $\sum_{j=0}^{n-1} 2^j + 2^j$
- $\sum_{j=0}^{n-1} 2^j + 2^{n-1}$
- $\sum_{j=0}^{n-1} 2^j + 2^n$
- $\sum_{j=0}^{n} 2^j + 2^n$

2) $\sum_{j=0}^{n+2} 2^{j-1}$

- $\sum_{j=0}^{n+1} 2^{j-1} + 2^n$
- $\sum_{j=0}^{n-1} 2^{j-1} + 2^{n+1}$
- $\sum_{j=0}^{n+1} 2^{j-1} + 2^{n+1}$

CHALLENGE ACTIVITY 7.3.1: Pulling out a final term from a summation.

Change of variables in summations

When a variable is used to denote the lower or upper limit of a sum, the value of the variable must be provided in order to evaluate the sum. In the summation below, the value of the sum depends on the variable n:

$$\sum_{j=1}^{n} (j+2)^3$$

By contrast, the variable j used for the index is internal to the sum and can be replaced with any other variable name:

$$\sum_{j=1}^{n}(j+2)^3 = \sum_{k=1}^{n}(k+2)^3$$

More complex substitutions can be done for the index variable which require that the upper and lower limit be adjusted. Consider the substitution i = k + 2. Here are the steps:

1. Determine the new lower limit: when k = 1, i = k + 2 = 1 + 2 = 3.
2. Determine the new upper limit: when k = n, i = k + 2 = n + 2.
3. Replace the variable in the terms: If i = k + 2, then k = i - 2. Replace k in the expression inside the sum with i - 2:

$$(k+2)^3 = ((i-2)+2)^3 = i^3.$$

Putting it all together, we get

$$\sum_{k=1}^{n}(k+2)^3 = \sum_{i=3}^{n+2} i^3$$

To verify that the two sums are the same, verify that the expanded form for both sums is:

$$3^3 + 4^3 + \cdots + (n+1)^3 + (n+2)^3$$

The animation below illustrates with another example:

PARTICIPATION ACTIVITY 7.3.4: Change of variables in summations.

Animation captions:

1. In the summation $\sum_{j=1}^{17} 2^{j-1}$, substitute $k = j - 1$.
2. Substitute the initial index $j = 1$ into the expression for k to get the initial condition for k: $k = j - 1 = 1 - 1 = 0$.
3. The new sum has an initial index of $k = 0$: $\sum_{k=0}$.
4. Substitute the final index $j = 17$ into the expression for k to get the final condition for k: $k = j - 1 = 17 - 1 = 16$. The new sum is now $\sum_{k=0}^{16}$.
5. The equality $k = j - 1$ implies $j = k + 1$. Substitute the expression for j in the summation term: $2^{j-1} = 2^{(k+1)-1} = 2^k$. The final sum is $\sum_{k=0}^{16} 2^k$.

PARTICIPATION ACTIVITY 7.3.5: Change of variables in summations.

Substitute the variable i with j = i + 2 in the summation below:

$$\sum_{i=3}^{21} \frac{1}{(i+3)} = \sum_{j=?}^{?} ?$$

1) What is the lower limit of the sum with index j?

Check Show answer

2) What is the upper limit of the sum with index j?

Check Show answer

3) Give an expression for i in terms of j.

i =

Check Show answer

4) The term in the summation is a fraction

$$\frac{1}{(i+3)} = \frac{1}{(?)}.$$

Give an expression for the missing denominator in terms of j.

Check Show answer

Closed forms for sums

A ***closed form*** for a sum is a mathematical expression that expresses the value of the sum without summation notation. There are closed form expressions for some, although not all, of the summations that arise naturally in scientific applications. For example, there is a closed form expression for the sum of terms in an arithmetic sequence:

Theorem 7.3.1: Closed form for the sum of terms in an arithmetic sequence.

For any integer n ≥ 1:

$$\sum_{k=0}^{n-1}(a+kd) = an + \frac{d(n-1)n}{2}$$

For example, in the sum 5 + 8 + 11 + 14 + 17 + 20 + 23, the initial term a = 5, the common difference d = 3, and the number of terms in the sum n = 7. The theorem says that:

$$5+8+11+14+17+20+23 = \sum_{j=0}^{6}(5+3j) = 5\cdot 7 + \frac{3\cdot 6\cdot 7}{2} = 98$$

There is also a closed form expression for the sum of terms in a geometric sequence:

Theorem 7.3.2: Closed form for the sum of terms in a geometric sequence.

For any real number r ≠ 1 and any integer n ≥ 1:

$$\sum_{k=0}^{n-1} a\cdot r^k = \frac{a(r^n-1)}{r-1}$$

For example, in the sum 3 + 6 + 12 + 24 + 48 + 96, the initial term a = 3, the common ratio r = 2, and the number of terms in the sum n = 6. The theorem says that:

$$3+6+12+24+48+96 = \sum_{j=0}^{5} 3\cdot 2^j = \frac{3(2^6-1)}{1} = 189$$

The two theorems above will be proven in the section on mathematical induction.

Example 7.3.1: Total sales of a company with exponential growth.

Consider a growing company whose sales are expected to grow at a rate of 10% each month. That is, if the company sells x widgets in one month, the company is expected to sell 1.1(x) widgets in the following month. If the company sells 1000 widgets in January, what can the company project its total sales to be for the entire year?

The sequence of number of widgets sold each month is a geometric sequence so the total sales can be obtained by using the closed form:

$$\sum_{k=0}^{n-1} a\cdot r^k = \frac{a(r^n-1)}{r-1}$$

The initial value a = 1000. The common ratio r = 1.1. The total number of months in a year is n = 12:

$$\sum_{k=0}^{11} 1000(1.1)^k = \frac{1000((1.1)^{12} - 1)}{1.1 - 1} \approx 21384$$

PARTICIPATION ACTIVITY 7.3.6: Computing sums of geometric and arithmetic sequences.

Give numerical values for the following summations.

1) $\sum_{k=0}^{199} k$

Check Show answer

2) $\sum_{k=0}^{9} 2^k$

Check Show answer

CHALLENGE ACTIVITY 7.3.2: Summations.

Note: You may want to use paper and pencil, and/or a calculator.

Additional exercises

Exercise 7.3.1: Evaluating summations.

About

Evaluate the following summations.

(a) $\sum_{k=-1}^{4} k^2$.

(b) $\sum_{k=0}^{4} 2^k$.

(c) $\sum_{k=2}^{} k3$

$\sum_{k=-3}^{\kappa} \cdot$

(d) $\sum_{k=0}^{3} 3^k$.

(e) $\sum_{k=0}^{200} (2 + 3k)$.

(f) $\sum_{k=0}^{200} 2 \cdot (1.01)^k$.

(g) $\sum_{k=0}^{100} (3 + 5k)$.

(h) $\sum_{k=0}^{100} 3 \cdot (1.1)^k$.

Exercise 7.3.2: Expressing sums using summation notation.

About

Express the following sums using summation notation.

(a) $(-2)^5 + (-1)^5 + \cdots + 7^5$

(b) $(-2) + (-1) + 0 + 1 + 2 + 3 + 4 + 5$

(c) $2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8$

(d) $0^3 + 1^3 + 2^3 + 3^3 + 4^3 + 5^3 + \cdots + 17^3$

(e) The sum of the cubes of the first 15 positive integers.

(f) The sum of the squares of the odd integers between 0 and 100.

Exercise 7.3.3: Pulling out a final term from a summation.

About

For each of the following expressions, write down an equivalent expression where the last term in the sum is outside the summation.

(a) $\sum_{j=-2}^{18} 2^j$.

(b) $\sum_{k=0}^{n}(k^2 - 4k + 1)$.

(c) $\sum_{k=0}^{m+2}(k^2 - 4k + 1)$.

Exercise 7.3.4: Variable substitution for indices of summations.

About

(a) Substitute variable i for j, where i = j + 2, in the summation below:

$$\sum_{j=0}^{n}(j + 2)$$

(b) Substitute variable j for k, where j = k - 1, in the summation below:

$$\sum_{k=0}^{n-1} 2^{k-2}$$

(c) Substitute variable k for j, where k = j - 4, in the summation below:

$$\sum_{j=4}^{17}(2j + 4)$$

Exercise 7.3.5: Are the two summations equal?

About

(a) Indicate whether the following equality is true and justify your answer:

$$\sum_{j=0}^{100} j^3 = \sum_{j=1}^{100} j^3$$

.

Exercise 7.3.6: Building a car collection.

About

A Silicon Valley billionaire purchases 3 new cars for his collection at the end of every month. Let a_n denote the number of cars he has after n months. Let $a_0 = 23$.

(a) What is a_8?

(b) If he pays $50 each month to have each car maintained, what is the total amount that he has paid for maintenance after 2 years? No need to calculate the actual number. Instead give a closed form (without the summation) mathematical expression for the number. Note that he purchases the new cars at the end of each month, so during the first month, he is only maintaining 23 cars.

Exercise 7.3.7: Raising rabbits.

About

A population of rabbits on a farm grows by 12% each year. Define a sequence $\{r_n\}$ describing the rabbit population at the end of each year. Suppose that the sequence starts with $r_0 = 30$.

(a) Give a mathematical expression for r_{12}. (You don't have to actually compute the number.)

(b) If each rabbit consumes 10 pounds of rabbit food each year, then how much rabbit food is consumed in 10 years? For simplicity, you can omit the food consumed by the baby rabbits born in a given year. For example, suppose the farm starts tabulating rabbit food on January 1, 2012 at which time the rabbit population is 30. You will count the food consumed by those 30 rabbits during 2012. You won't count the food consumed by the rabbits born in 2012 until after January 1, 2013. Again, you don't have to compute the number, but you do have to give a closed form (without the summation) mathematical expression for the number.

7.4 Mathematical induction

Suppose that one day a genie grants you three wishes. You make two wishes and then for your third wish, you wish for three more wishes the next day. Given the fact that you can always use your third wish to renew your wishes for the next day, it is possible to prove that from that first day onward, you can have three wishes every day for the rest of your life. This is an example of the principle of mathematical induction (or just *induction* for short).

Induction is a proof technique that is especially useful for proving statements about elements in a sequence. An inductive proof establishes that some statement parameterized by n is true, for any positive integer n. In the example with the three wishes, the sequence is the sequence of days after that first day of wishes. The fact that is being proven is that you are able to have three wishes on the n^{th} day for n = 1, 2, ...

Figure 7.4.1: The two components of an inductive proof.

- The ***base case*** establishes that the theorem is true for the first value in the sequence.
  - The genie grants you three wishes on day 1
- The ***inductive step*** establishes that if the theorem is true for k, then the theorem also holds for k + 1.
  - If you have three wishes on day k, then you can get three wishes for day k+1 (by using the third wish to ask for three more wishes the next day).

The ***principle of mathematical induction*** states that if the base case (for n = 1) is true and inductive step is true, then the theorem holds for all positive integers.

Figure 7.4.2: Principle of mathematical induction.

Let S(n) be a statement parameterized by a positive integer n. Then S(n) is true for all positive integers n, if:

1. S(1) is true (the base case).
2. For all $k \in \mathbf{Z}^+$, S(k) implies S(k+1) (the inductive step).

The inductive step is a compact way to state that an infinite sequence of implications are true:

For all $k \in \mathbf{Z}^+$, S(k) implies S(k+1) $\leftrightarrow$ [S(1) implies S(2)] and [S(2) implies S(3)] and [S(3) implies S(4)] and ...

In the statement "S(k) implies S(k+1)" of the inductive step, the supposition that S(k) is true is called the ***inductive hypothesis***. The animation below illustrates why the principle of induction works by showing how the two components of an inductive proof imply that S(n) is true for any positive integer n:

PARTICIPATION ACTIVITY 7.4.1: Why induction works.

Animation content:

undefined

Animation captions:

1. $P(1)$ is true, and for all $k \geq 1$, $P(k)$ implies $P(k + 1)$. Setting $k = 1$ means that $P(1)$ implies $P(2)$.
2. Therefore, P(2) is also true.
3. For $k = 2$, $P(2)$ implies $P(3)$. Therefore, $P(3)$ is also true. For $k = 3$, $P(3)$ implies $P(4)$. Therefore, $P(4)$ is also true.
4. Since P(k) implies P(k + 1) for all k ≥ 1,the process can be continued up to any n ≥ 1
5. Therefore, for all $n \geq 1$, $P(n)$ is true.

PARTICIPATION ACTIVITY 7.4.2: The components of an inductive proof.

Theorem: For all n ∈ $\mathbf{Z}^+$, Q(n) is true,

where Q(n) is a statement that is parameterized by a positive integer n.

1) In an inductive proof of the theorem, what must be proven in the base case?
   - Q(0) is true
   - Q(1) is true
   - Q(k) is true
   - Q(n) is true

2) In an inductive proof of the theorem, what must be proven in the inductive step?
   - For all positive integers k, Q(k - 1) implies Q(k)
   - For all positive integers k, Q(k) implies Q(n)
   - For all positive integers k, Q(k) implies Q(k + 1)
   - For all positive integers k, Q(k)

3) If the inductive step says that for all positive integers Q(k) implies Q(k+1), then what is the inductive hypothesis?
   - Q(k+1)
   -

Q(n)

Q(k)

The next example will be a proof that for any positive integer n, the following equality is true:

$$P(n): \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

If n is assigned a particular value, then the values of both sides of the equation are determined. For example, if n = 5, the equality P(5) is obtained by replacing n with 5 in P(n):

$$P(5): \sum_{j=1}^{5} j = \frac{5(5+1)}{2}.$$

The left side of the equation is the sum 1 + 2 + 3 + 4 + 5. The expression on the right side of the equation is 5(5 + 1)/2. Now it is possible to check that both sides evaluate to the same number (in this case 15). The assertion that the equality holds for a particular n can be denoted by the predicate P(n). The statement P(n) may be true or false, depending on the value of n.

PARTICIPATION ACTIVITY 7.4.3: The components of an inductive proof.

Define the predicate P(n) to be the statement:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

Theorem: For every positive integer n, P(n) is true.

1) $P(n): \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

Express the statement P(3).

- $1 + 2 + \ldots + n = \frac{3(3+1)}{2}$
- $1 + 2 + 3 + \ldots + j = \frac{3(3+1)}{2}$
- $1 + 2 + 3 = \frac{3(3+1)}{2}$
- $1 + 2 + 3 = \frac{n(n+1)}{3}$

2) The theorem states that for every positive integer n, P(n) is true. In an inductive proof of the theorem, what would be proven for the base case?

- $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$
- $\sum_{j=1}^{0} j = \frac{0(0+1)}{2}$
- $\sum_{j=1}^{1} j = \frac{1(1+1)}{2}$

3) $P(n): \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

What is P(k + 1)?

- $\sum_{j=1}^{k+1} j = \frac{(k+1)(k+2)}{2}$
- $\sum_{j=1}^{k+1} j = \frac{k(k+1)}{2}$
- $\sum_{j=1}^{k+1} j = \frac{n(n+1)}{2}$

4) $P(n): \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

In an inductive proof of the theorem, what would be proven in the inductive step?

- For any integer k ≥ 1, $\sum_{j=1}^{k+1} j = \frac{(k+1)(k+2)}{2}$
- For any integer k ≥ 1, if $\sum_{j=1}^{k} j = \frac{k(k+1)}{2}$ then $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$
- For any integer k ≥ 1, if $\sum_{j=1}^{k} j = \frac{k(k+1)}{2}$ then $\sum_{j=1}^{k+1} j = \frac{(k+1)(k+2)}{2}$

The complete proof by induction is given below for the theorem that P(n) is true for all positive integers n. The proof begins with the statement "By induction on n", which lets the reader know that the proof will be an inductive proof and the variable n is the parameter to which the principle of induction is applied. In the beginning of the inductive step, the proof states explicitly: "Suppose that for positive integer k, P(k) is true, then we will show that P(k + 1) is also true." The statement gives the reader an overview of the coming argument. It is also important to point out at which step the inductive hypothesis is being applied.

Example 7.4.1: Proving an identity by induction.

Theorem: For every positive integer n, \sum_{j=1}^n j = \frac{n(n+1)}{2}

Proof.
By induction on n.

Base case: n = 1.

When n = 1, the left side of the equation is \sum_{j=1}^1 j = 1.

When n = 1, the right side of the equation is 1(1 + 1)/2 = 1.

Therefore, \sum_{j=1}^1 j = \frac{1(1+1)}{2} .

Inductive step: Suppose that for positive integer k, \sum_{j=1}^k j = \frac{k(k+1)}{2} , then we will show that \sum_{j=1}^{k+1} j = \frac{(k+1)(k+2)}{2}

Starting with the left side of the equation to be proven:

\begin{align} \sum_{j=1}^{k+1} j &= \sum_{j=1}^{k} j + (k+1)~~~~~~~~&~\mbox{by separating out the last term}\\ \\ &= \frac{k(k+1)}{2} + (k+1)&~\mbox{by the inductive hypothesis}\\ \\ &= \frac{k(k+1) + 2(k+1)}{2}&~\\ \\ &= \frac{(k+2)(k+1)}{2}&~\mbox{by algebra}\\ \\ \end{align}
Therefore, \sum_{j=1}^{k+1} j = \frac{(k+1)(k+2)}{2} . ■

Video explaining the inductive step (2:14).

Sometimes the smallest n for which S(n) holds is some constant c other than 1. Then the base case must show that S(c) is true. The inductive step will establish that S(k)→S(k+1) for any k ≥ c. An inductive proof assumes that S(k) and k ≥ c are true and uses those two facts to establish that S(k+1) is true.

For example, let S(n) be the inequality $2^n \geq 3n$. S(3) is not true because $2^3 = 8$ which is not greater than or equal to 3 · 3 = 9. The next inductive proof will show that for n ≥ 4, S(n) is true.

Example 7.4.2: Proving an inequality by induction.

Theorem: For n ≥ 4, $2^n \geq 3n$.

Proof.
By induction on n.

Base case: n = 4.

2^4 = 16 \ge 12 = 3 \cdot 4
Therefore, for n = 4, $2^n \geq 3n$.

Inductive step: We will show that for any integer k ≥ 4, if $2^k \geq 3k$, then

$$2^{k+1} \geq 3(k + 1).$$

Starting with the left side of the inequality to be proven:

\begin{align} 2^{k+1}& = 2 \cdot 2^k~~~~~~~~&~\mbox{by algebra}\\ \\ &\ge 2 \cdot 3k&~\mbox{by the inductive hypothesis}\\ \\ & = 3k + 3k&~\\ \\ &\geq 3k +

3&~\mbox{because $k \geq 4 \geq 1$}\\ \\ & = 3(k+1)&~\mbox{by algebra}\\ \\ \end{align}

Therefore, $2^{k+1} \geq 3(k+1)$ ■.

Video explaining the inductive step (3:42).

PARTICIPATION ACTIVITY 7.4.4: Proving an inequality by induction.

Consider a proof of the following fact:

For all $n \geq 4$, $2^n \geq n^2$

1) What should be proven in the base case?
   - For $n = 1$, $2^n \geq n^2$
   - For $n = 4$, $2^n \geq n^2$
   - $2^k \geq k^2$
   - For every $k \geq 4$, if $2^k \geq k^2$, then $2^{k+1} \geq (k+1)^2$

2) What should be proven in the inductive step?
   - For $n = 4$, $2^n \geq n^2$
   - $2^k \geq k^2$
   - For every $k \geq 4$, if $2^k \geq k^2$, then $2^{k+1} \geq (k+1)^2$

3) Below is an argument for the inductive step. In which line is the inductive hypothesis used?

\begin{align} 2^{k+1} &= 2 \cdot 2^k & (1)\\ \\ & \ge 2 \cdot k^2 & (2)\\ \\ & = k^2 + k^2 & (3)\\ \\ & \ge k^2 + 4k & ~\mbox{because $k \ge 4$}\\ \\ & = k^2 + 2k + 2k & (4)\\ \\ & \ge k^2 + 2k +1 = (k+1)^2 & \end{align}

   - Line 1
   - Line 2
   - Line 3
   - Line 4

Additional exercises

Exercise 7.4.1: Components of an inductive proof.

About

Define P(n) to be the assertion that:

$\sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6}$

(a) Verify that P(3) is true.

(b) Express P(k).

(c) Express P(k + 1).

(d) In an inductive proof that for every positive integer n, $\sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6}$ what must be proven in the base case?

(e) In an inductive proof that for every positive integer n, $\sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6}$ what must be proven in the inductive step?

(f) What would be the inductive hypothesis in the inductive step from your previous answer?

(g) Prove by induction that for any positive integer n, $\sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6}$

Exercise 7.4.2: Proving identities by induction.

About

Prove each of the following statements using mathematical induction.

(a) Prove that for any positive integer n, $\sum_{j=1}^nj^3 = \left(\frac{n(n+1)}2\right)^2$

(b) Prove that for any positive integer n, $\sum_{j=1}^nj\cdot 2^j = (n-1)2^{n+1}+2$

(c) Prove that for any positive integer n, \sum_{j=1}^nj(j-1) = \frac{n(n^2-1)}3

Exercise 7.4.3: Proving inequalities by induction.

About

Prove each of the following statements using mathematical induction.

(a) Prove that for $n \geq 2$, $3^n > 2^n + n^2$.

(b) For any $n \geq 1$, the factorial function, denoted by n!, is the product of all the positive integers through n:
n! = 1\cdot 2 \cdot 3 \cdots (n - 1)\cdot n
Prove that for $n \geq 4$, $n! \geq 2^n$.

(c) Prove that for $n \geq 1$, \sum_{j = 1}^n \frac{1}{j^2} ≤ 2 - \frac 1 n

7.5 More inductive proofs

Divisibility proof by induction

The next example of a proof by induction proves that 3 evenly divides the value of a mathematical expression parameterized by an integer n. A quantity is evenly divisible by 3 if it can be expressed as 3·m for some integer m. Define the predicate Q(n) to be the statement:

Q(n): 3 evenly divides $2^{2n} - 1$.

The statement Q(3) says that 3 evenly divides $2^{2 \cdot 3} - 1$. Q(3) is true because $2^{2 \cdot 3} - 1 = 2^6 - 1 = 63$ and 3 does in fact evenly divide 63 (because 63 = 3·21).

The following example shows an inductive proof of the theorem that Q(n) is true for all positive integers n.

Example 7.5.1: Proving a divisibility proof by induction.

Theorem: For every positive integer n, 3 evenly divides $2^{2n} - 1$.

Proof.
By induction on n.

Base case: n = 1.

$2^{2 \cdot 1} - 1 = 2^2 - 1 = 3$. Since 3 evenly divides 3, the theorem holds for the case n = 1.

Inductive step: Suppose that for positive integer k, 3 evenly divides $2^{2k} - 1$. Then we will show that 3 evenly divides $2^{2(k+1)} - 1$.

By the inductive hypothesis, 3 evenly divides $2^{2k} - 1$, which means that $2^{2k} - 1 = 3m$ for some integer m. By adding 1 to both sides of the equation $3m = 2^{2k} - 1$, we get $2^{2k} = 3m + 1$ which is an equivalent statement of the inductive hypothesis.

We must show that $2^{2(k+1)} - 1$ can be expressed as 3 times an integer.

$$\begin{aligned} 2^{2(k+1)}-1 &= 2^{2k+2} - 1 \\ &= 4 \cdot 2^{2k} - 1 && \text{by algebra} \\ &= 4 (3m +1) - 1 && \text{by the inductive hypothesis} \\ &= 3 \cdot 4m + 4 - 1 \\ &= 3(4m + 1) && \text{by algebra} \end{aligned}$$

Since m is an integer, (4m + 1) is also an integer. Therefore $2^{2(k+1)} - 1$ is equal to 3 times an integer which means that $2^{2(k+1)} - 1$ is divisible by 3. ■

PARTICIPATION ACTIVITY 7.5.1: Divisibility proof by induction.

Theorem: For every positive integer n, 4 evenly divides $3^{2n} - 1$.

1) Suppose that an inductive proof of the theorem uses the following inductive hypothesis: For positive integer k, 4 evenly divides $3^{2k} - 1$. Which statement is equivalent to the inductive hypothesis?
   - For some integer m, $4m = 3^{2k}$.
   - For some integer m, $3m = 3^{2k} - 1$.
   - For some integer m, $4m + 1 = 3^{2k}$.

2) In the inductive step, the proof assumes that the inductive hypothesis is true and must prove that 4 evenly divides $3^{2(k+1)} - 1$. Which expression implies that 4 evenly divides $3^{2(k+1)} - 1$?
   - $3^{2(k+1)} - 1 = 4(9m + 2)$, for some integer m.
   - $3^{2(k+1)} = 4(9m + 2)$, for some integer m.
   -

$3^{2(k+1)} - 1 = 4(9m + 1/2)$, for some integer m.

Using induction to prove an assertion about a recurrence relation

Induction is particularly well suited to proving facts about sequences defined by recurrence relations. The theorem below starts by defining a sequence with a recurrence relation and initial value. The theorem asserts that a particular explicit formula (given in the last line of the theorem statement) corresponds to the same sequence defined by the recurrence relation.

Theorem 7.5.1: Explicit formula for a sequence defined by a recurrence relation.

Define the sequence $\{g_n\}$ as:

- $g_0 = 1$.
- $g_n = 3 \cdot g_{n-1} + 2n$, for any $n \geq 1$.

Then for any $n \geq 0$, g_n = \frac 5 2 \cdot 3^n - n - \frac 3 2

The recurrence relation ($g_n = 3 \cdot g_{n-1} + 2n$) and initial value are given as part of the assumptions of the theorem and can be used at any point in the proof. In the inductive step of the proof, we assume the explicit formula is correct for term g_k and then prove that it is also correct for term g_{k+1}. The inductive step will therefore need to express g_{k+1} in terms of g_k. However, the recurrence relation defines g_n in terms of g_{n-1}. The proof, therefore, uses a slightly different form of the recurrence relation which is obtained by replacing the index n with k + 1. Note that with the change of variable, the first index for the recurrence is k = 0 instead of n = 1. The two statements of the recurrence relation are equivalent: \begin{align} g_n & = 3 g_{n-1} + 2n~~~&\mbox{for}~~n\ge 1\\ \\ g_{k+1} & = 3 g_{k} + 2(k+1)~~~&\mbox{for}~~k\ge 0\\ \end{align} Both statements are equivalent to the infinite sequence of equations:

- $g_1 = 3 \cdot g_0 + 2 \cdot 1$
- $g_2 = 3 \cdot g_1 + 2 \cdot 2$
- $g_3 = 3 \cdot g_2 + 2 \cdot 3$
- etc.

Example 7.5.2: Proving the explicit formula for a recurrence relation by induction.

Proof.
By induction on n.

Base case: n = 0.

We must show that g_0 = \frac 5 2 \cdot 3^0 - 0 - \frac 3 2. Since \frac 5 2 \cdot 3^0 - 0 - \frac 3 2 = 1 and the initial condition states that $g_0 = 1$, the theorem holds for n = 0.

Inductive step: Suppose that for any integer k ≥ 0, g_k = \frac 5 2 \cdot 3^k - k - \frac 3 2. Then we will show that g_{k+1} = \frac 5 2 \cdot 3^{k+1} - (k+1) - \frac 3 2

For any integer k ≥ 0:

\begin{align} g_{k+1} &= 3 g_{k} + 2(k+1)~~~~~~~~&~\mbox{by definition}\\ \\ &= 3 \left(\frac 5 2 3^k - k - \frac 3 2 \right) + 2(k+1)~~~~~~~~&~\mbox{by the inductive hypothesis}\\ \\ & = \frac 5 2 \cdot 3 \cdot 3^k - 3k - 3 \cdot \frac 3 2 +2k + 2&\\ \\ &= \frac 5 2 3^{k+1} - k - \frac 5 2&\\ \\ & = \frac 5 2 3^{k+1} - (k+1) - \frac 3 2&~\mbox{by algebra}\\ \\ \end{align}
Therefore, g_{k+1} = \frac 5 2 \cdot 3^{k+1} - (k+1) - \frac 3 2 . ■

The next set of questions present an inductive proof of the following theorem:

Theorem 7.5.2: Explicit formula for a sequence defined by a recurrence relation.

Define the sequence $\{h_n\}$ as:

- $h_0 = 7$
- $h_n = (h_{n-1})^3$, for any $n \geq 1$

Then for any n ≥ 0, h_n = 7^{(3^n)}

PARTICIPATION ACTIVITY 7.5.2: Using induction to prove an explicit formula for a sequence defined by a recurrence relation.

1) Select the choice that is equivalent to the recurrence relation: h_n = (h_{n-1})^3~~~~~\mbox{for}~n \ge 1

   - h_k = (h_{k-1})^3~~~~~\mbox{for}~k \ge 0
   - h_{k+1} = (h_{k})^3~~~~~\mbox{for}~k \ge 0
   - h_{k+1} = (h_{k-1})^3~~~~~\mbox{for}~k \ge

0

2) What would be proven in an inductive step of the proof?

- If for $k \geq 0$, $h_k = (h_{k-1})^3$, then h_k = 7^{(3^k)}
- If for $k \geq 0$, h_{k} = 7^{(3^{k})} , then $h_{k+1} = (h_k)^3$
- If for $k \geq 0$, h_k = 7^{(3^k)} then h_{k+1} = 7^{(3^{k+1})}

3) The inductive proof starts out with: \begin{align} h_{k+1} &= (?)~~&~\mbox{by definition}\\ \\ &= \left(7^{(3^k)} \right)^3 &~\mbox{by the ind. hyp.}\\ \end{align} Select the correct term for (?).

- \left(h_k \right)^3
- 7^{(3^{k+1})}
- 7^{(3 \cdot 3^{k})}

Inductive proofs showing the closed forms for sums of arithmetic and geometric sequences.

The theorem below gives a closed form for the sum of the first n terms in an arithmetic sequence. Here we present an inductive proof of the theorem.

Theorem 7.5.3: Closed form for the sum of terms in an arithmetic sequence.

For any integer $n \geq 1$:

\sum_{j=0}^{n-1} (a + jd) = an + \frac{d(n-1)n}{2}

Proof: By induction on n.

The base case is proven for n = 1. In this case, the summation goes from j = 0 to n - 1 = 0. Therefore, there is only one term in the summation in which j = 0. The value of the summation for n = 1 is $a + 0 \cdot d = a$. Plugging in n = 1 into the expression on the right side of the equality also yields a value of a.

The inductive step assumes the theorem holds for positive integer k (the inductive hypothesis):

\sum_{j=0}^{k-1} (a + jd) = ak + \frac{d(k-1)k}{2}

and uses the inductive hypothesis to show that the theorem holds for k+1:

\sum_{j=0}^{k} (a + jd) = a(k+1) + \frac{dk(k+1)}{2}

The remainder of the proof is given in the following animation:

PARTICIPATION ACTIVITY 7.5.3: The inductive step in the proof of the closed form for the sum of an arithmetic sequence.

Animation captions:

1. Start with the left hand side of the fact to be proven and separate the last term from the summation: \sum_{j=0}^{k}(a + jd) = (a + kd) + \sum_{j=0}^{k-1}(a + jd).
2. Apply the inductive hypothesis by replacing \sum_{j=0}^{k}(a + jd) with ka + \frac {dk(k-1)} {2} to get (a + kd) + ka + \frac{dk(k-1)}{2}.
3. Rearrange the expression using algebra to get a + ka + \frac{2kd + dk(k-1)}{2}.
4. Pull out common factors from terms to get a(k+1) + \frac{d(2k + k^2 - k)}{2}
5. Simplifying further gives a(k+1) + \frac{dk(k+1)}{2} which is equal to the initial expression \sum_{j=0}^{k}(a+jd).

There is also a theorem giving an closed form expression for the sum of terms in a geometric sequence:

Theorem 7.5.4: Closed form for the sum of terms in a geometric sequence.

For any real number r ≠ 1 and any integer n ≥ 1:

\sum_{j=0}^{n-1} a\cdot r^j = \frac{a(r^n-1)}{r-1}

An inductive proof of the theorem is given in the animation below:

PARTICIPATION ACTIVITY 7.5.4: An inductive proof of the closed form for the sum of a geometric sequence.

Animation content:

undefined

Animation captions:

1. The base case proves the theorem for n = 1.\sum_{k=0}^{0} ar^k = ar^0 = a. Also, \frac{a(r^1 - 1)}{r-1} = a. The two expressions are equal.
2. The inductive step assumes that \sum_{k=0}^{n-2} ar^k = \frac{a(r^{n-1}-1)}{r-1} .
3. The inductive step proves that \sum_{k=0}^{n-1} ar^k = \frac{a(r^{n}-1)}{r-1} .
4. The inductive step starts with the left side of the equality to be proven and pulls out the last term from the summation: \sum_{k=0}^{n-1} ar^k = ar^{n-1} + \sum_{k=0}^{n-2} ar^k .

5. The inductive hypothesis is applied by replacing \sum_{k=0}^{n-2} ar^k with \frac{a(r^{n-1}-1)}{r-1} to get ar^{n-1} + \frac{a(r^{n-1}-1)}{r-1} .
6. The fractions are combined and simplified to get \frac{a[r^n - r^{n-1}+r^{n-1}-1]}{r-1}.
7. The two middle terms in the numerator cancel, resulting in \frac{a[r^n-1]}{r-1}, which is equal to the initial expression \sum_{k=0}^{n-1} ar^k .

PARTICIPATION ACTIVITY 7.5.5: Induction proofs for sums of arithmetic and geometric sequences.

1) In the proof of the closed form for the sum of an arithmetic sequence, the inductive hypothesis is: \sum_{j=0}^{k-1} (a + jd) = ka + \frac{ak(k-1)}{2}. Select the expression that is equivalent to the expression below by applying the inductive hypothesis: (a + kd) + \sum_{j=0}^{k-1} (a + jd)
   - (a + kd) + ka + \frac{ak(k-1)}{2}
   - (a + kd) + ka + \frac{ak(k+1)}{2}
   - \sum_{j=0}^{k} (a + jd)

2) In the inductive proof of the sum of a geometric sequence, the inductive hypothesis is that for positive integer k, \sum_{j=0}^{k-1} ar^j = \frac{a(r^k-1)}{1-r}. What must be proven assuming the inductive hypothesis is true?
   - \sum_{j=0}^{k-1} ar^j = \frac{a(r^k-1)}{1-r}.
   - \sum_{j=0}^{k} ar^j = (ar^k)+ \sum_{j=0}^{k-1} ar^j
   - \sum_{j=0}^{k} ar^j = \frac{a(r^{k+1}-1)}{1-r}.

An inductive proof for set operations

Induction is a useful technique for proving theorems about other mathematical objects, beyond sequences and sums. The animation below uses induction to prove a generalized version of de Morgan's law for sets. The theorem assumes that de Morgan's law is true for two sets and shows that de Morgan's law can be extended to an arbitrary number of sets.

PARTICIPATION ACTIVITY 7.5.6: Inductive proof of the generalized de Morgan's law for sets.

Animation captions:

1. Inductive proof of a Generalized De Morgan's law for sets. The base case for n = 2 is just De Morgan's law: $\overline{A_1 \cap A_2} = \overline{A_1} \cup \overline{A_2}$.
2. The inductive step assumes the theorem for k: $\overline{A_1 \cap A_2 \cap \ldots \cap A_k} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_k}$ and proves the theorem for k+1.
3. B is defined to be $A_1 \cap A_2 \cap \ldots \cap A_k$. Therefore, $\overline{A_1 \cap A_2 \cap \ldots \cap A_k \cap A_{k+1}} = \overline{B \cap A_{k+1}}$.
4. $\overline{B \cap A_{k+1}} = \overline{B} \cup \overline{A_{k+1}}$ by De Morgan's law.
5. $\overline{B} \cup \overline{A_{k+1}} = \overline{A_1 \cap A_2 \cap \ldots \cap A_k} \cup \overline{A_{k+1}}$ by the definition of B.
6. The result of applying the inductive hypothesis to the last expression is $\overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_k}$, which is equal to the first expression $\overline{A_1 \cap A_2 \cap \ldots \cap A_k}$.

PARTICIPATION ACTIVITY 7.5.7: Induction proof for generalized de Morgan's law for sets.

1) What is the inductive hypothesis in the inductive proof for the generalized de Morgan's law for sets?

- $\overline{A_1 \cap A_2 \cap \cdots \cap A_{k-1}} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_{k-1}}$
- $\overline{A_1 \cap A_2 \cap \cdots \cap A_k} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_k}$
- $\overline{A_1 \cap A_2 \cap \cdots \cap A_{k+1}} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_{k+1}}$

Additional exercises

Exercise 7.5.1: Proving divisibility results by induction.

About

Prove each of the following statements using mathematical

induction.

(a) Prove that for any positive integer n, 4 evenly divides 3^{2n}-1.

(b) Prove that for any positive integer n, 6 evenly divides 7^n - 1.

Exercise 7.5.2: Proving explicit formulas for recurrence relations by induction.

About

Prove each of the following statements using mathematical induction.

(a) Define the sequence $\{c_n\}$ as follows:
- $c_0 = 5$
- $c_k = (c_{k-1})^2$ for $k \geq 1$

Prove that for $n \geq 0$, $c_n = 5^{2^n}$.
Note that in the explicit formula for c_n, the exponent of 5 is 2^n.

(b) Define the sequence $\{b_n\}$ as follows:
- $b_0 = 1$
- $b_k = 2b_{k-1} + 1$ for $k \geq 1$

Prove that for $n \geq 0$, $b_n = 2^{n+1} - 1$.

7.6 Strong induction and well-ordering

The Fibonacci sequence is defined by initial values $f_0 = 0$, $f_1 = 1$ and the recurrence relation for $n \geq 2$:

$$f_n = f_{n-1} + f_{n-2}$$

Suppose we wanted to prove an upper bound on the Fibonacci numbers, such as $f_n \leq 2^n$, for all $n \geq 0$. In the standard form of induction, the inductive step would use the fact that $f_k \leq 2^k$ to argue that $f_{k+1} \leq 2^{k+1}$. However, since f_{k+1} is defined in terms of f_k and f_{k-1}, the proof requires a bound on both f_k and f_{k-1} in order to upper bound f_{k+1}. The inductive hypothesis for standard induction only provides the bound on f_k.

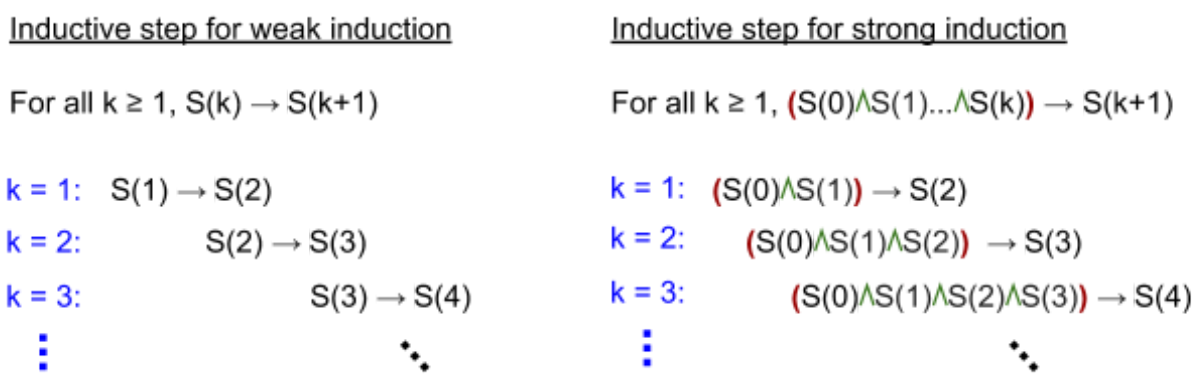
For the upper bound on terms of the Fibonacci sequence, a stronger version of induction is required. The ***principle of strong induction*** assumes that the fact to be proven holds for all values less than or equal to k and proves that the fact holds for k+1. By contrast the standard form of induction only assumes that the fact holds for k in proving that it holds for k+1. Define S(n) to be the assertion that $f_n \leq 2^n$, where f_n is defined by Fibonacci's initial values and recurrence relation. The proof that S(n) is true for all non-negative integers n has two components:

- **Base case:** S(0) and S(1) are true.
- **Inductive step:** For every k ≥ 1, (S(0) ∧ S(1) ∧ .... ∧ S(k)) implies S(k + 1).

In strong induction the inductive hypothesis is S(0) ∧ S(1) ∧ .... ∧ S(k). In regular (or weak) induction, the inductive hypothesis is just S(k). Sometimes the inductive hypothesis is expressed using a universal quantifier. The following two ways to express the inductive hypothesis are equivalent:

S(0) ∧ S(1) ∧ .... ∧ S(k) is true. ↔ For every j in the range 0 through k, S(j) is true.

Figure 7.6.1: Expanded expression of the inductive step for weak and strong induction.



PARTICIPATION ACTIVITY 7.6.1: Using strong induction.

Suppose that the inductive step of a strong induction proof shows that for k ≥ 1, if (S(0) ∧ S(1) ∧ .... ∧ S(k)) is true, then S(k+1) is true.

1) What statement is equivalent to the inductive hypothesis?
   - ○ For k ≥ 1 and any j in the range from 0 through k, S(j) is true.
   - ○ For k ≥ 1 and any j in the range from 0 through k+1, S(j) is true.
   - ○ S(j) is true.

2) If you assume that the inductive hypothesis is true, can you conclude that S(k-2) is true?
   - ○ Yes
   - ○ No

3) If you assume that the inductive hypothesis is true, can you conclude

that S(k-1) is true?

- Yes
- No

There is no simple rule to determine when strong induction is required instead of the standard form of induction. The best method is to try proving the inductive step using only the preceding value (i.e., try and prove the statement for k + 1 using only the statement for k). If that fails, then try a stronger inductive hypothesis. The inductive proof showing a bound on the Fibonacci numbers given below would fail with standard induction because bounds on f_k and f_{k-1} are required to prove the bound for f_{k+1}.

Example 7.6.1: Upper bound for Fibonacci numbers proven using strong induction.

Theorem: For $n \geq 0$, $f_n \leq 2^n$.

Proof.
By strong induction on n.

Base case:

- n = 0: $f_0 = 0 \leq 2^0$
- n = 1: $f_1 = 1 \leq 2^1$

Inductive step:

For $k \geq 1$, suppose that for any j in the range from 0 through k, $f_j \leq 2^j$. We will prove that $f_{k+1} \leq 2^{k+1}$.

Since $k \geq 1$, then $k-1 \geq 0$. Therefore both k and k-1 fall in the range from 0 through k, and by the inductive hypothesis, $f_{k-1} \leq 2^{k-1}$ and $f_k \leq 2^k$.

$$\begin{aligned} f_{k+1} &= f_k + f_{k-1} && \text{by definition} \\ &\leq 2^k + 2^{k-1} && \text{by the inductive hypothesis} \\ &\leq 2^k + 2^{k-1} + 2^{k-1} && \text{since } 2^{k-1} \geq 0 \\ &= 2^k + 2 \cdot 2^{k-1} \\ &= 2^k + 2^k = 2 \cdot 2^k = 2^{k+1} && \text{by algebra} \end{aligned}$$

Therefore, $f_{k+1} \leq 2^{k+1}$. ■

Generalized strong induction: multiple base cases

The predicate S(n) may be true only for n greater than or equal to a particular value denoted below by the constant a. Also, the base case may require proving the assertion directly for more than one value. The ***base case*** for a proof by strong induction establishes that S(n) holds for n = a through b, where a and b are constants. The ***inductive step*** in a proof by strong induction assumes that S(j) is true for all values of j in the range from a through some integer $k \geq b$ and then proves that theorem holds for k+1.

Figure 7.6.2: Principle of strong mathematical induction.

Let S(n) be a statement parameterized by n, a positive integer. Let a ≤ b be integers. Then S(n) is true for all n ≥ a, if the following two conditions hold:

1. S(a), S(a+1), ..., S(b) are true (the base case).
2. For all k ≥ b, if (S(a) ∧ S(a+1) ∧ .... ∧ S(k)) is true, then S(k+1) is also true (the inductive step).

Figure 7.6.3: Expanded expression of the inductive step with an arbitrary range for the base case.

Here is the inductive step in expanded form. The base case must establish that the inductive hypothesis is true for the smallest value for k.

Base case: $(S(a) \wedge S(a+1) \wedge .. \wedge S(b))$

Inductive step: For all k ≥ b, $(S(a) \wedge S(a+1) ... \wedge S(k)) \rightarrow S(k+1)$

k = b: $(S(a) \wedge S(a+1) \wedge .. \wedge S(b)) \rightarrow S(b+1)$

k = b+1: $(S(a) \wedge S(a+1) \wedge .. \wedge S(b) \wedge S(b+1)) \rightarrow S(b+2)$

k = b+2: $(S(a) \wedge S(a+1) \wedge .. \wedge S(b) \wedge S(b+1) \wedge S(b+2)) \rightarrow S(b+3)$

⋮ ⋱

PARTICIPATION ACTIVITY 7.6.2: The principle of strong induction with general range for the base case.

1) Consider an inductive proof in which the inductive step is:

   For all k ≥ 6, if P(3), P(4),...,P(k) are all true, then P(k+1) is true.

   What facts must be proven in the base case?

   - ○ P(3) only.
   - ○

P(3), P(4), and P(5).

- ○ P(3), P(4), P(5), and P(6).

2) Select the choice that is equivalent to the following statement:

For k ≥ 7, if P(j) is true for any j in the range 2 through k, then P(k+1) is true.

- ○ For k ≥ 7, (P(2) ∧ P(3) ∧...P(j)) implies P(j+1)
- ○ For k ≥ 7, (P(2) ∧ P(3) ∧...P(k)) implies P(k+1)
- ○ For k ≥ 7, (P(7) ∧ P(8) ∧...P(k)) implies P(k+1)

Strong induction example: buying cans of juice in 3-packs or 4-packs.

Suppose that cans of juice come in packs of 3 or 4. We would like to be able to buy n cans of juice by purchasing a combination of 3-packs or 4-packs. For which values of n is this possible? Let P(n) be the assertion that it is possible to buy n cans of juice by purchasing only 3-packs or 4-packs. P(7) is true because we can purchase one 3-pack and one 4-pack. P(5) is not true because there is no way to combine 3-packs and 4-packs to get five cans of juice. We will use strong induction to prove that for any n ≥ 6, P(n) is true. Here is an outline of the proof:

- **Base case:** Prove P(6), P(7), and P(8) directly.
- **Inductive step:** For k ≥ 8, assume that that P(j) is true for any j in the range 6 through k, and prove that P(k + 1) is true.
  - Argue that P(k - 2) is true by showing that k - 2 falls in the range 6 through k, covered in the inductive hypothesis. The argument uses the fact that k ≥ 8 and therefore k - 2 ≥ 6.
  - Since P(k - 2) is true, k-2 cans of juice can be bought by combining 3-packs and 4-packs. By adding one 3-pack to the k - 2 cans of juice, there will be k - 2 + 3 = k + 1 cans of juice. Therefore P(k + 1) is true.

PARTICIPATION ACTIVITY 7.6.3: Inductive proof for buying n cans of juice.

Animation captions:

1. The base case starts with n = 6. Purchasing two 3-packs of juice results in 6 cans. Therefore, P(6) is true.
2. n = 7 is next. Purchasing one 3-pack and one 4-pack results in 7 cans. Therefore, P(7) is true.
3. n = 8 is next. Purchasing two 4-packs results in 8 cans. Therefore, P(8) is true.

4. The inductive hypothesis is that for k \ge 8, P(j) is true for any j in the range 6 through k.
5. The inductive step proves P(k+1). Start with k-2 cans of juice.
6. Since k \ge 8, then k−2 \ge 6. Therefore, k-2 is in the range 6 through k and by the inductive hypothesis, k-2 cans of juice can be purchased.
7. When an additional 3-pack is purchased, the number of cans is (k-2) + 3 = k+1. Therefore, P(k+1) is true.

How many base cases?

How did we know to use three base cases in the example with the cans of 3-packs and 4-packs of juice? To some extent, determining how many base cases are required in a strong induction proof requires some trial and error, but there are some basic principles that can be applied. In proving that P(k + 1) is true, we needed to assume that P(k - 2) true in order to add one 3-pack to get k +1 cans of juice. If the base cases had only included P(6) and P(7), then we would have been unable to prove the inductive step for k = 7: (P(6) ∧ P(7)) → P(8). In proving P(8), we needed to assume P(8 - 3) = P(5), but P(5) is not included in the base cases. Note that, the inductive step could have added a 4-pack at the end instead of a 3-pack but using a 4-pack would have required 4 consecutive bases cases (6 through 9) instead of 3 consecutive base cases.

PARTICIPATION ACTIVITY 7.6.4: Strong induction.

Consider again the problem of purchasing cans of juice. Suppose now that juice is sold in 2-packs or 5-packs. Let Q(n) be the statement that it is possible to buy n cans of juice by combining 2-packs and 5-packs. This problem discusses using strong induction to prove that for any n ≥ 4, Q(n) is true.

1) The inductive step will prove that for k ≥ 5, Q(k+1) is true. What is the inductive hypothesis?
   - ○ Q(j) is true for any j = 4, 5, ..., k.
   - ○ Q(j) is true for any j = 4, 5, ..., k+1.
   - ○ Q(j) is true for any j = 5, ..., k+1.

2) The inductive step will prove that for k ≥ 5, Q(k+1) is true. What part of the inductive hypothesis is used in the proof?
   - ○ Q(k-2)
   - ○ Q(k-1)
   - ○ Q(k)

3) In the base case, for which values of n

should Q(n) be proven directly?

- n = 3, n = 4, and n = 5.
- n = 4 and n = 5.
- n = 4, n = 5, and n = 6.

The next example comes from number theory. The assertion is that every positive integer greater than 1 can be written as a product of prime numbers. For example: $\begin{align} 20 & = 2 \cdot 2 \cdot 5\\ 156 & = 2 \cdot 2 \cdot 3 \cdot 13\\ 71 & = 71 \end{align}$

The last example, 71, is itself prime so it is the product of only one prime number.

Example 7.6.2: Using strong induction to prove a theorem in number theory.

Theorem: For any integer n ≥ 2, n can be expressed as a product of prime numbers.

Proof.
By strong induction on n.

Base case: n = 2. Since 2 is a prime number, it already is a product of one prime number: 2.

Inductive step: Assume that for k ≥ 2, any integer j in the range from 2 through k can be expressed as a product of prime numbers. We will show that k + 1 can be expressed as a product of prime numbers.

If k + 1 is prime, then it is a product of one prime number, k + 1. If k + 1 is not prime, k + 1 is composite and can be expressed as the product of two integers, a and b, that are each at least 2. We need to show that both a and b are at most k in order to apply in the inductive hypothesis.

Since k+1 = a·b, then a = (k+1)/b. Furthermore, since b ≥ 2, then a = (k+1)/b < k+1. If a is an integer which is strictly less than k + 1, then a ≤ k. The symmetric argument can be used to show that b = (k+1)/a ≤ k. Thus a and b both fall in the range from 2 through k which means that the inductive hypothesis can then be applied and they can each be expressed as a product of primes:

$\begin{align} a & = p_1 \cdot p_2 \cdots p_l\\ b & = q_1 \cdot q_2 \cdots q_m \end{align}$

Now k + 1 can be expressed as a product of primes:

$k+1 = a \cdot b = (p_1 \cdot p_2 \cdots p_l) \cdot (q_1 \cdot q_2 \cdots q_m)$.

■

PARTICIPATION ACTIVITY 7.6.5: Applying the inductive hypothesis.

1) In the proof that every integer greater

than 1 can be expressed as a product of primes, it is necessary to apply the inductive hypothesis to integers a and b. What must be proven about a and b in order to apply the inductive hypothesis?

- a and b are greater than 1.
- a and b are in the range from 2 through k + 1.
- a and b are in the range from 2 through k.

The principle of induction is closely related to another principle regarding the integers called the well-ordering principle:

Figure 7.6.4: The well-ordering principle.

The ***well-ordering principle*** says that any non-empty subset of the non-negative integers has a smallest element.

It can be shown that the principle of mathematical induction, the principle of strong induction, and the well-ordering principle are all equivalent. In other words, using any one of the principles as an assumption, the other two principles can be proven. Below is a proof showing that if the well-ordering principle is true, then the principle of mathematical induction is true. The principle of induction is stated again below for review:

Figure 7.6.5: The principle of mathematical induction.

Let P(n) be a predicate that is parameterized by non-negative integers n. If the following two conditions hold:

- **Base case:** P(1) is true
- **Inductive step:** For all k ≥ 1, P(k) implies P(k+1)

then for all n ≥ 1, P(n) is true.

Figure 7.6.6: Proof that well-ordering implies the principle of mathematical induction

mathematical induction.

Proof.
Proof by contrapositive: We will show that if the well-ordering principle is true and S(n) is false for some n ≥ 1, then at least one of the two conditions for induction must fail.

Suppose that S(n) is false for some n ≥ 1. Consider the set of integers n such that n ≥ 1 and S(n) is not true. By the well-ordering principle, there must be a smallest element m in that set. Note that m ≠ 0 because 0 is not a member of the set. (One of the criteria for an integer n to be included in the set is that n ≥ 1.) If m = 1, then S(1) is not true and the base case fails. If m ≥ 2, then S(m - 1) is true but S(m) is false. Letting k = m - 1, k ≥ 1 and S(k) is true but S(k + 1) is not true. Therefore, the inductive step fails for some k ≥ 1. Thus, if the statement S(n) is false for some n ≥ 1, then one of the two conditions for the principle of mathematical induction must be false. ■

The well-ordering principle can be used to prove some facts directly. Typically, a proof that uses the well-ordering principle is a proof by contradiction. There is a statement S(n), parameterized by an integer n and the assertion to be proven is that S(n) is true for all n ≥ b for some constant b. Define a set that consists of all integers which violate the assertion to be proven. The set contains all integers n such that n ≥ b and S(n) is false. The well-ordering principle says that if the set is non-empty then it must have a smallest element. The details of the proof of course depend on the nature of the statement S(n), but typically a contradiction arises by examining the smallest element in the set. Often the contradiction involves showing that there must be a yet a smaller value for which S(n) is false. If the set of all integers n ≥ b such that S(n) is false can not have a smallest element, the set must be empty and therefore S(n) must be true for all n ≥ b. We illustrate the use of the well-ordering principle in proving the following theorem:

Theorem 7.6.1: Quotient and remainder theorem.

Let n be an integer and d a positive integer. There are integers q and r such that n = qd + r and 0 ≤ r < d.

The number q is the integer quotient when n is divided by d and r is the remainder. It turns out that given n and d, the numbers q and r are unique. For now, the well-ordering principle will be used simply to show that they exist.

Figure 7.6.7: Proof of the quotient and remainder theorem using the well-ordering principle.

Proof.

Consider the set S of all non-negative integers of the form (n - td), where t is an integer. The set is non-empty because t can be chosen so that -td is as large as desired and large enough to make (n - td) non-negative. Let r be the smallest number in the set S and let q be the value for t which achieves the minimum: r = n - qd. By definition, r is non-negative and therefore r ≥ 0. The fact that r < d will be proven by contradiction. Suppose that r ≥ d. Then r - d ≥ 0.

r-d = (n-qd)-d = n-(q+1)d

Since r - d can be written in the form (n - td) by selecting t = q + 1, r was not the smallest non-negative number in the set of integers of the form n - td. A contradiction. ■

PARTICIPATION ACTIVITY 7.6.6: Using the well-ordering principle.

The well-ordering principle can be used to prove any fact which can be proven by standard or strong induction.

Consider again the problem of purchasing cans of juice. Suppose that juice is sold in 2-packs or 5-packs. Let Q(n) be the statement that it is possible to buy n cans of juice by combining 2-packs and 5-packs. We can use the principle of well-ordering to show that for n ≥ 4, Q(n) is true.

1) To which set should the well-ordering principle be applied?
   - ○ The set of all n ≥ 1 such that Q(n) is false.
   - ○ The set of all n ≥ 4 such that Q(n) is true.
   - ○ The set of all n ≥ 4 such that Q(n) is false.

2) Let m be the smallest value of the set from question 1. Select the statement that correctly describes m.
   - ○ Q(m) is false, but for any j in the range from 4 through m - 1, Q(j) is true.
   - ○ For any j in the range from 4 through m, Q(j) is true.
   - ○ Q(m) is true, but for any j in the range from 4 through m - 1, Q(j) is false.

3) What is the contradiction that results if m = 4 or m = 5?
   - ○ Since it is possible to buy 2 or 3

cans, just add an extra 2-pack to get 4 or 5 cans.

- It is possible to buy 4 cans (two 2-packs) and 5 cans (one 5-pack). Therefore, Q(4) and Q(5) are both true.

4) What is the contradiction that results if $m \geq 6$?

- It is possible to buy m - 2 cans. m cans can be purchased by buying m - 2 cans and adding a 2-pack.
- Q(6) is true because it is possible to buy 6 cans by purchasing three 2-packs.

Additional exercises

Exercise 7.6.1: Proofs by strong induction - combining stamps.

About

Prove each of the following statements using strong induction.

(a) Prove that any amount of postage worth 8 cents or more can be made from 3-cent or 5-cent stamps.

(b) Prove that any amount of postage worth 24 cents or more can be made from 7-cent or 5-cent stamps.

Exercise 7.6.2: Proofs by strong induction - explicit formulas for recurrence relations.

About

Prove each of the following statements using strong induction.

(a) The Fibonacci sequence is defined as follows:

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$, for $n \geq 2$

Prove that for $n \geq 0$,

f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}2 \right)^n - \left(\frac{1 - \sqrt{5}}2 \right)^n \right]

(b) Define the sequence $\{h_n\}$ as follows:

- $h_0 = 5/3$
- $h_1 = 11/3$
- $h_n = 3h_{n-1} + 4h_{n-2} + 6n$, for $n \geq 2$

Prove that for $n \geq 0$,
h_n = 2 \cdot 4^n + \frac 3 2 (-1)^n - n - \frac{11}{6}

(c) Define the sequence $\{g_n\}$ as follows:

- $g_0 = 51$
- $g_1 = 348$
- $g_n = 5g_{n-1} - 6g_{n-2} + 20 \cdot 7^n$ for $n \geq 2$

Prove that for $n \geq 0$, $g_n = 2^n + 3^n + 7^{n+2}$.

Exercise 7.6.3: Proof by the well-ordering principle.

About

Prove each of the following statements using the principle of well-ordering.

(a) Define the sequence $\{g_n\}$ as follows:

- $g_0 = 51$
- $g_1 = 348$
- $g_n = 5g_{n-1} - 6g_{n-2} + 20 \cdot 7^n$ for $k \geq 2$

Prove that for $n \geq 0$, $g_n = 2^n + 3^n + 7^{n+2}$.

(b) Prove that any amount of postage worth 8 cents or more can be made from 3-cent or 5-cent stamps.

7.7 Loop invariants

This section has been set as optional by your instructor.

Program correctness

The field of ***program verification*** is concerned with formally proving that programs perform correctly. A program's correct behavior is defined by stating that if a ***pre-condition*** is true before the program starts, then the program will end after a finite number of steps and a ***post-condition*** is true after the program ends. Ex: If a program must sort 3 numbers like (50, 22, 93):

- Pre-condition: The input is a sequence of three numbers.
- Post-condition: The output is a reordering of the three numbers so that each number is less than or equal to the next number in the sequence. For input (50, 22, 93), the correct output would be (22, 50, 93).

Table 7.7.1: Examples of program pre-conditions and post-conditions.

Program description	Pre-condition	Post-condition
Sort a list of numbers.	Input is a positive integer n, and a list of n numbers a_1, \ldots, a_n.	Output is b_1, b_2, \ldots, b_n, a reordering of a_1, \ldots, a_n such that b_j \le b_{j+1}, for every j \in \{1, \ldots, n-1\}.
Find a given number in a list of numbers.	Input is a positive integer n, a number x, and a list of n numbers a_1, \ldots, a_n.	Output is an integer m such that 1 \le m \le n and a_m = x or -1 if x \ne a_j for every j \in \{1, \ldots, n\}
Compute the square root of a non-negative real number.	Input is a real number x such that x \ge 0.	Output is a real number y such that y \ge 0 and y^2 = x.

PARTICIPATION ACTIVITY 7.7.1: Selecting a pre-condition for a program.

Match each program description to the best pre-condition.

Determine whether a sequence of numbers is increasing

Multiply two numbers

Compute the log base 2 of a number

Compute the cube root of a number

	The input is a real number x
	The input variables x and y are two real numbers
	The input variables are n, a positive integer, and a sequence of n numbers a_1, \ldots, a_n.
	The input is a real number x such that x \gt 0.

Reset

PARTICIPATION ACTIVITY 7.7.2: Selecting a post-condition for a program.

Match each program description to the best post-condition. The pre-condition for each program is that the input is a positive integer n and a list of n numbers a_1, \ldots, a_n.

Find a maximum number in a list of numbers

Compute the sum of a list of numbers

Find a minimum number in a list of numbers

Compute the average of a list of numbers

	m is an integer such that 1 \le m \le n and a_m \ge a_j for every j \in \{1, \ldots, n\}.
	m is an integer such that 1 \le m \le n and a_m \le a_j for every j \in \{1, \ldots, n\}.
	A number x such that \sum_{j=1}^n a_j.
	A number x such that x = \frac 1 n \sum_{j=1}^n a_j.

Reset

A program analyst can show that a large program performs correctly by breaking the program into smaller segments, each segment having a pre-condition and post-condition. The post-condition for one segment is the pre-condition for the next segment. Then, for each segment, the analyst must *prove* that if the pre-condition for that segment is true before the segment executes, then the post-condition for the segment is true after the segment executes.

Figure 7.7.1: Pre-conditions and post-conditions for a program that computes the minimum of three values.

```
ComputeMin(x, y, z)

    // Pre-condition for the program = pre-condition for Segment 1
    // [x, y, z are three numbers]

min := x;  //Segment 1

    // Post-condition for Segment 1 = pre-condition for Segment 2
    // [min \in \{x\} and min \le x]

If (y \le min), then min := y;  //Segment 2

    // Post-condition for Segment 2 = pre-condition for Segment 3
    // [min \in \{x, y\}, min \le x, and min \le y]

If (z \le min), then min := z;  //Segment 3

    // Post-condition for Segment 3 = post-condition for the program
    // [min \in \{x, y, z\}, min \le x, min \le y, and min \le z]

Return(min)
```


PARTICIPATION ACTIVITY 7.7.3: Pre-conditions and post-conditions for a program that computes the sum of three numbers.

```
ComputeSum(x, y, z)

    // [Assertion 1]  Pre-condition for the program = pre-condition for Segment 1

sum := 0   //Segment 1

    // [Assertion 2]   Post-condition for Segment 1 = pre-condition for Segment 2

sum := sum + x   //Segment 2

    // [Assertion 3]   Post-condition for Segment 2 = pre-condition for Segment 3

sum := sum + y   //Segment 3

    // [Assertion 4]   Post-condition for Segment 3 = pre-condition for Segment 4

sum := sum + z   //Segment 4

    // [Assertion 5]   Post-condition for Segment 4 = post-condition for the program

Return(sum)
```


Select the statement that corresponds to each assertion.

sum = x | **sum = x + y + z** | **sum = 0** | **x, y, and z are numbers** | **sum = x + y**

	Assertion 1
	Assertion 2
	Assertion 3
	Assertion 4
	Assertion 5

Reset

Loop invariants for while loops

This material focuses on showing that a program segment consisting of a while-loop performs correctly. A while loop has the form shown below, where C is a loop condition that evaluates to true

or false. If the ***loop condition*** for a while loop is true, the instructions inside the loop are executed. Otherwise, the loop terminates and the next statement after the while loop is executed.

```
While (C)
  [List of instructions]
End-while
```

A ***loop invariant*** is an assertion that is true before each iteration of a loop. The animation below illustrates the use of a loop invariant to establish that the given while-loop correctly computes x^n. There are four steps in using a loop invariant to show that a while loop performs correctly.

Figure 7.7.2: Steps in using a loop invariant.

Given a while loop with condition C and a loop invariant I, the four steps below are sufficient to establish that if the pre-condition is true before the loop, then the post-condition is true after the loop:

1. Show that if the pre-condition is true before the loop begins, then I is also true.
2. Show that if C and I are both true before an iteration of the loop, then I is true after the iteration.
3. Show that the condition C will eventually be false.
4. Show that if \negC and I are both true, then the post-condition is true.

PARTICIPATION ACTIVITY 7.7.4: The four steps of using a loop invariant.

Animation captions:

1. Pre-condition is that n is a non-negative integer, j = 0, and power = 1. Post-condition says that power = x^n. The loop invariant is that j is an integer j \le n, and power = x^j.
2. Step 1 assumes the pre-condition and proves the loop invariant. For the power function, we assume n is non-negative, j = 0, and power = 1 and prove that j is an integer, j \le n, and power = x^j.
3. Step 2. A subscript 1 denotes values of variables j and power before an iteration and subscript 2 denotes values after the iteration.
4. Assume j_1 \lt n, j_1 is an integer, and power = x^{j_1}. Prove that j_2 is an integer such that j_2 \le n, and power_2 = x^{j_2}.
5. Step 3 shows that the loop condition j \lt n will eventually be false.
6. Step 4 shows that if the loop condition is false and the loop invariant is true then the post-condition will be true.

PARTICIPATION ACTIVITY 7.7.5: Completing the proof using a loop invariant for the power function: Step 1.

The loop to compute x^n is given below:

```
While (j \lt n)
  power := power * x
  j := j + 1
End-while
```

Step 1 shows that if the pre-condition is true, then the loop invariant is true before the first iteration of the loop.

Pre-condition: n is a non-negative integer, j = 0, and power = 1
Loop invariant: j is an integer such that j \le n and power = x^j.

1) Step 1, part A.
   Assume: n is a non-negative integer, j = 0, and power = 1
   Prove: power = x^j

   x^j = x^0 = 1 = power.

   Which of the three equalities does *not* rely on the assumption?

   - ○ x^j = x^0
   - ○ x^0 = 1
   - ○ 1 = power

2) Step 1, part B.
   Assume: n is a non-negative integer, j = 0, and power = 1
   Prove: j is an integer and j \le n.

   (Line 1) If n is non-negative then j = 0 \le n
   (Line 2) Since j = 0, then j \le n
   (Line 3) Since j = 0, then j is an integer

   Which fact is *not* used in any of the lines of the proof?

   - ○ n is an integer
   - ○ n is non-negative
   - ○ j = 0

PARTICIPATION ACTIVITY 7.7.6: Completing the proof using a loop condition for the power function: Step 2.

The loop to compute x^n is given below:

```
While (j \lt n)
  power := power * x
  j := j + 1
End-while
```

Step 2 shows that if the loop condition and the loop invariant are both true before an iteration of the loop, then the loop invariant is true after the iteration of the loop.

Variables j and power are the two variables whose value changes during an iteration of the loop. j_1 and power_1 will denote the values before the iteration, and j_2 and power_2 will denote the values after the iteration.

Loop condition: j_1 \lt n
Loop invariant before the iteration: j_1 is an integer, j_1 \le n, and power_1 = x^{j_1}
Loop invariant after the iteration: j_2 is an integer, j_2 \le n, and power_2 = x^{j_2}.

1) What is the correct relationship between j_1 and j_2?

   - j_2 = j_1 + 1
   - j_2 = j_1
   - j_1 = j_2 + 1

2) What is the correct relationship between power_1 and power_2?

   - power_2 = power_1 \cdot x
   - power_1 = power_2 \cdot x.
   - power_2 = x^{j_2}

3) Step 2, part A.
   Assume: j_1 \lt n, j_1 is an integer, and power_1 = x^{j_1}.
   Prove: power_2 = x^{j_2}.

   power_2 = power_1 \cdot x = x^{j_1}\cdot x = x^{j_1+1} = x^{j_2}

   Which fact in the assumptions is used in the argument?

   - j_1 \lt n
   - j_1 is an integer
   - power_1 = x^{j_1}

4) Step 2, part B.
   Assume: j_1 \lt n, j_1 is an integer, and

power_1 = x^{j_1}.
Prove: j_2 is an integer and j_2 \le n.

(Line 1) Since j_2 = j_1 + 1 and j_1 is an integer, then j_2 is also an integer.
(Line 2) If j_1 is an integer and j_1 \lt n, then j_1 \le n-1.
(Line 3) Since j_1 \le n-1, then j_1 +1 \le n.
(Line 4) Since j_2 = j_1 + 1 and j_1 +1 \le n, then j_2 \le n.

Which fact in the assumptions is not used in the argument?

- j_1 \lt n
- j_1 is an integer
- power_1 = x^{j_1}

PARTICIPATION ACTIVITY 7.7.7: Completing the proof using a loop condition for the power function: Steps 3 and 4.

The loop to compute x^n is given below:

```
While (j \lt n)
  power := power * x
  j := j + 1
End-while
```

Step 3 shows that the loop condition will eventually be false. Step 4 shows that if the loop condition is false and the loop invariant is true then the post-condition is true.

Loop condition: j \lt n
Loop invariant: j is an integer, j \le n, and power = x^{j}
Post-condition: power = x^n.

1) Step 3.
Prove: The condition j \lt n will eventually be false.

Which fact can be used to justify that fact j \lt n will eventually be false?

- After n iterations, power = x^n,
- After n-1 iterations, j = n-1
- After n iterations, j = n.

2) Step 4.
Assume: j is an integer, j \le n, power = x^j, and \neg(j \lt n).
Prove: power = x^n

(Line 1) Since j \lt n is false, then j \ge n.
(Line 2) Since j \ge n and j \le n, then j = n.
(Line 3) Since j = n and power = x^j, then power = x^n.

Which fact in the assumptions is not used in the argument?

- ○ j is an integer
- ○ j \le n
- ○ power = x^j
- ○ \neg(j \lt n)

Here are all the steps put together showing that the loop correctly computes the x^n.

```
While (j \lt n)
  power := power * x
  j := j + 1
End-while
```

- Pre-condition: n is a non-negative integer, j = 0, and power = 1
- Post-condition: power = x^n
- Loop invariant: j is an integer such that j \le n and power = x^j.

Figure 7.7.3: A complete proof using a loop invariant.

Step 1. Assume that n is a non-negative integer, j = 0, and power = 1. We will prove that that j is an integer such that j \le n and power = x^j.

Since j = 0 and power = 1, x^j = x^0 = 1 = power. Since n is a non-negative integer, n \ge 0 = j. Also, since j = 0, j is an integer.

Step 2. Let j_1 and power_1 denote the values of j and power before an iteration of the loop. Let j_2 and power_2 denote the values of j and power after the iteration. Assume that j_1 \lt n, j_1 is an integer, and power_1 = x^{j_1}. Prove that j_2 is an integer such that j_2 \le n, and power_2 = x^{j_2}.

The first line of the loop multiplies power by x, so power_2 = power_1 \cdot x. The second line increments j by 1, so j_2 = j_1 + 1.

Since j_1 is an integer and j_1 \lt n, then j_1 \le n-1. Adding 1 to both sides of the inequality yields that j_1 + 1 \le n, which means that j_2 \le n. Also, power_2 = power_1 \cdot x = x^{j_1} \cdot x = x^{j_1 +1} = x^{j_2} Finally, since j_1 is an integer, then j_2 is also an integer.

Step 3. Since the value of j is n after n iterations of the loop, the condition j \lt n will be false after n iterations. Therefore the loop will eventually terminate.

Step 4. Assume that j \le n, power = x^j, and the condition j \lt n is false. We will prove that power = x^n.

Since j \lt n is false, then j \ge n. If j \ge n and j \le n, then j = n. Therefore power = x^j = x^n.

PARTICIPATION ACTIVITY 7.7.8: Identifying the steps in using a loop invariant.

The loop below computes the sum of a list of numbers.

```
While (j \lt n)
   sum := sum + a_{j+1}
   j := j + 1
End-while
```

- Pre-condition: j = 1, sum = a_1, n is a positive integer, a_1, \ldots, a_n is a list of n numbers.
- Post-condition: sum = \sum_{k=1}^n a_k.
- Loop invariant: j is an integer, j \le n, and sum = \sum_{k = 1}^j a_k

Fill in the blanks to express what must be proven in each step. j_1 and sum_1 denote the values of j and sum before an iteration of the loop, and j_2 and sum_2 denote the values after the iteration.

1) Step 1.
   Assume that n is a positive integer, j = 1, and sum = a_1.
   Prove (?).
   - ○ sum = \sum_{k=1}^n a_k
   - ○ j is an integer, j \le n, and sum = \sum_{k = 1}^j a_k.
   - ○ j \lt n

2) Step 2. Assume that (?), j_1 is an integer, and sum_1 = \sum_{k = 1}^{j_1}a_k.
   Prove that j_2 is an integer, j_2 \le n, and sum_2 = \sum_{k = 1}^{j_2}a_k.

- j_1 \lt n
- j_2 \lt n
- j_1 \ge n

3) Step 3. After a finite number of iterations (?).

- j \lt n
- j \le n
- j \ge n

4) Step 4. Assume that (?), j \le n, and sum = \sum_{k=1}^j a_k.
Prove that sum = \sum_{k=1}^n a_k.

- j \lt n
- j \le n
- j \ge n

Additional exercises

Exercise 7.7.1: Proving the correctness of while loops using loop invariants.

About

For each while loop, use the loop invariant given to show that if the pre-condition is true before the loop then the post-condition is true after the loop.In each step, clearly state what facts are assumed and what facts will be proven.

(a) The loop below computes the product of two non-negative integers.

```
While (j \lt n)
  prod := prod + m
  j := j + 1
End-while
```

- Pre-condition: m and n are non-negative integers. j = 0 and prod = 0.
- Post-condition: prod = m \cdot n
- Loop invariant: j is an integer, j \le n, and prod = m \cdot j

(b) The loop below computes the sum of a list of numbers.


```
While (j \lt n)
  sum := sum + a_{j+1}
  j := j + 1
End-while
```

- Pre-condition: j = 1, sum = a_1, n is a positive integer, a_1, \ldots, a_n is a list of n numbers.
- Post-condition: sum = \sum_{k=1}^n a_k.

- Loop invariant: j is an integer, j \le n, and sum = \sum_{k = 1}^j a_k

(c)

The loop below finds a maximum value in a list of numbers.

```
While (j \lt n)
  If (a_{max} \lt a_{j+1}), then max := j+1
  j := j + 1
End-while
```


- Pre-condition: j = 1, max = 1, n is a positive integer, a_1, \ldots, a_n is a list of n numbers.
- Post-condition: 1 \le max \le n and a_{k} \le a_{max} for every k \in \{1, \ldots, n\}.
- Loop invariant: j and max are integers such that 1 \le max \le j \le n. Also, for every k \in \{1, \ldots, j\}, a_{k} \le a_{max}.

7.8 Recursive definitions

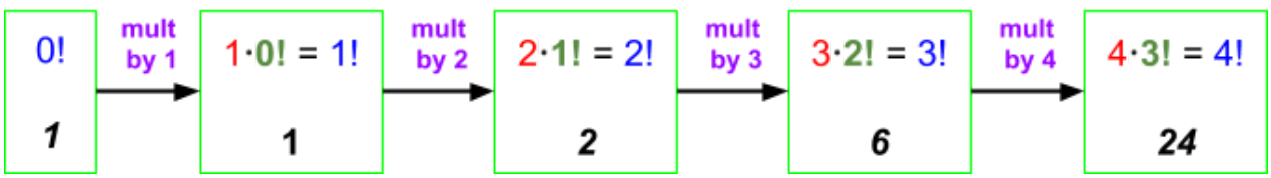
This section has been set as optional by your instructor.

The ***factorial*** function f(n) = n! for n ≥ 0 can be defined as: f(n) = n! = n \cdot (n-1) \cdots 1

A disadvantage of the definition given above is that the reader is required to infer what goes between the (n - 1) and the 1 terms. Also, the value of 0! is not clear from the definition. A more precise definition is n! = f(n) such that: \begin{align} f(0) & = 1\\ f(n) & = n\cdot f(n-1) ~~~~~~\mbox{for}~~ n \ge 1 \end{align}

The second definition is an example of a recursive definition. In a ***recursive definition*** of a function, the value of the function is defined in terms of the output value of the function on smaller input values. One can use the recursive definition for n! to determine the value of the factorial function on a particular value for n by starting at 0, multiplying 0! by 1 to get the value of 1! then multiplying by 2 to get 2!, and so on, until the desired n! has been reached. The process is called recursion.
Recursion is the process of computing the value of a function using the result of the function on smaller input values.

Figure 7.8.1: Using the recursive definition of n! to compute 4!



We can use recursion to define functions, sequences and sets. For example, a recurrence relation (along with initial values) is a recursive definition of a sequence because the recurrence relation shows how to compute the value of a term as a function of terms with smaller indices (i.e., terms that occur earlier in the sequence). A sequence is just special kind of function in which the domain is a consecutive set of integers. The notation for a sequence is slightly different since the input is specified with a subscript: $f_n = f(n)$. Here is a recurrence relation for sequence $\{f_n\}$ that is equivalent to the recursive definition for the factorial function: \begin{align} f_0 &= 1\\ f_n &= n\cdot f_{n-1} ~~~~~~\mbox{for}~~ n \ge 1\\ \end{align}

PARTICIPATION ACTIVITY 7.8.1: Computing the value of functions defined recursively.

1) Given the definition of a function g whose domain is the set of all non-negative integers:

\begin{align} g(0) &= 0\\ g(n) &= g(n-1) + n^3 ~~~\mbox{for}~~ n \ge 1 \end{align}
What is g(3)?

Check Show answer

2) Given the definition of a function h whose domain is the set of all non-negative integers:

\begin{align} h(0) &= 1\\ h(n) &= (n^2+1)\cdot h(n-1) ~~~\mbox{for}~~ n \ge 1 \end{align}
What is h(2)?

Check Show answer

PARTICIPATION ACTIVITY 7.8.2: Finding recursive definitions for functions on non-negative integers.

Find the recursive definition that matches the non-recursive definition given for each function.

1) $g(n) = \sum_{j=0}^n j^2$

- \begin{align} g(0) &= 0\\ g(n) &= n^2 + (g(n-1))^2

~~~\mbox{for}~~ n \ge 1
\end{align}

- \begin{align} g(0) & = 0\\ g(n) & = n^2 + g(n-1)
~~~\mbox{for}~~ n \ge 1
\end{align}
- \begin{align} g(0) & = 0\\ g(n) & = n^2 \cdot g(n-1)
~~~\mbox{for}~~ n \ge 1
\end{align}

2) h(n) = \left(n! \right)^2

- \begin{align} h(0) & = 1\\ h(n) & = n^2 \cdot h(n-1)
~~~\mbox{for}~~ n \ge 1
\end{align}
- \begin{align} h(0) & = 1\\ h(n) & = n^2 \cdot (h(n-1))^2
~~~\mbox{for}~~ n \ge 1
\end{align}
- \begin{align} h(0) & = 1\\ h(n) & = n \cdot (h(n-1))^2
~~~\mbox{for}~~ n \ge 1
\end{align}

Recursively defined sets.

Certain kinds of sets are most naturally specified with recursive definitions. A recursive definition of a set shows how to construct elements in the set by putting together smaller elements.

Definition 7.8.1: Components of a recursive definition of a set.

- A ***basis*** explicitly states that one or more specific elements are in the set.
- A ***recursive rule*** shows how to construct larger elements in the set from elements already known to be in the set. (There is often more than one recursive rule).
- An ***exclusion statement*** states that an element is in the set only if it is given in the basis or can be constructed by applying the recursive rules repeatedly to elements given in the basis.

Consider the task of specifying the correct syntax of a programming language, such as formally specifying that the parentheses within a statement must be properly nested. For simplicity, we disregard the rest of the characters in the statement and just focus on strings of left and right parentheses. For example, the string (() ()) is properly nested, but the string ( )) is not. Formally specifying which strings are and are not properly nested is important because a compiler must systematically check that an expression is in the correct format, and must otherwise issue an appropriate error statement. Below is a recursive definition for the set of all strings of parentheses that are properly nested:

Figure 7.8.2: Recursive definition of the set of all properly nested parentheses.

- Basis: The sequence () is properly nested.
- Recursive rules: If u and v are properly-nested sequences of parentheses then:
    1. (u) is properly nested.
    2. uv is properly nested.
- Exclusion statement: a string is properly nested only if it is given in the basis or can be constructed by applying the recursive rules to strings in the basis.

Since the exclusion statement is essentially the same for every recursively defined set, it is often excluded (but implied) in a recursive definition of a set.

The basis states that the string ( ) is properly nested. Rule 1 can be applied to ( ) to show that (( )) is also properly nested. Rule 2 can be applied by taking two copies of ( ) and concatenating them to get the string ( ) ( ) which is also properly nested. Properly nested strings of length 6 can be obtained by taking either one of the properly nested string of length 4 and applying rule 1: (( ) ( )) and ((( ))). In addition, a properly nested string of length 2 can be concatenated with a properly nested string of length 4 to get properly nested strings of length 6: ( )(( )), (( )) ( ), ( ) ( ) ( ).

Notice that in this definition, there is more than one way to apply the recursive rules to get the same string. For example, the string ( ) ( ) ( ) can be obtained by concatenating ( ) and ( ) ( ) in either order: ( ) ( ) ( ) or ( ) ( ) ( ). In some situations, it is important to have a set of recursive rules in which there is only one way to construct each element in the set using the recursive rules.

PARTICIPATION ACTIVITY 7.8.3: Constructing a properly nested string or parentheses recursively.

Animation captions:

1. The string ((())())(()) is properly nested because ((())())(()) can be constructed by repeatedly applying the recursive rules to the string () in the basis.
2. The string (()())) is not properly nested. Once the string (()()) is constructed, a right parenthesis cannot be added to the end because there is no matching left parenthesis.

PARTICIPATION ACTIVITY 7.8.4: Identifying properly nested parentheses.

Which strings of parentheses are properly nested?

1) (( ))

- Properly nested
- Not properly nested

2) (( )))

- Properly nested
- Not properly nested

3) ((( )) ( ))

- Properly nested
- Not properly nested

PARTICIPATION ACTIVITY 7.8.5: Applying recursive rules to get properly nested parentheses.

What was the last recursive rule applied in constructing each of the following properly nested string or parentheses?

Recursive rules:

1. (u) is properly nested.
2. uv is properly nested.

1) (( ) ( ))

- Rule 1
- Rule 2

2) (( ) ( )) (( ))

- Rule 1
- Rule 2

3) ((( ) ( )) (( )))

- Rule 1
- Rule 2

Example: binary strings

The set B^k, where B = {0, 1}, is defined to be the set of all binary strings of length k. The set of all binary strings without any restriction on length (denoted by B^*) is an infinite set. The ***empty string*** (denoted by the symbol λ) is the unique string whose length is 0. Since B^0 is the set of all binary strings of length 0, $B^0 = \{\lambda\}$. B^* includes all of B^k for any $k \geq 0$. One way to define B^* is by an infinite union:

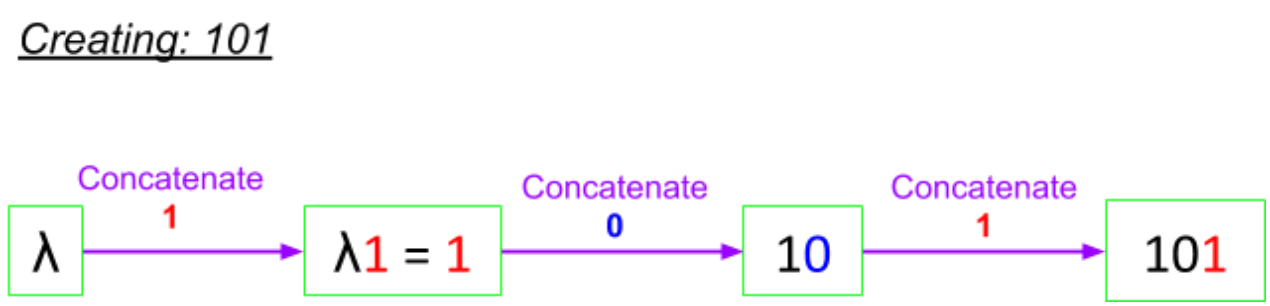
B^* = B^0 \cup B^1 \cup B^2 \cup \cdots

The recursive definition of B^* does not require using an infinite union of finite sets. The recursive rule constructs a longer string from a shorter string by concatenating 0 or 1. The recursive definition of B^* is:

- Base case: $\lambda \in B^*$
- Recursive rule: if $x \in B^*$ then,
  - $x0 \in B^*$
  - $x1 \in B^*$

Every binary string can be constructed by starting with λ and repeatedly concatenating 0 or 1 to the end of the string. The diagram below shows an example:

Figure 7.8.3: Constructing a binary string using the recursive definition.



PARTICIPATION ACTIVITY 7.8.6: How many applications of the recursive rule to build a string?.

1) How many applications of the recursive rule given in the definition of B^* are required to construct the string 1?

Check Show answer

2) How many applications of the recursive rule given in the definition of B^* are required to construct the string 11001?

Check Show answer

Recursive definition for the length of a binary string

The length |x| of a binary string x is a function that maps every binary string to a non-negative integer. The length of a binary string can be defined recursively based on the recursive definition of $\{0,1\}^*$.

- Base case: $|\lambda| = 0$
- Recursive rule: if $x \in B^*$ then,
  - $|x0| = |x|+1$
  - $|x1| = |x|+1$

PARTICIPATION ACTIVITY 7.8.7: Applying the recursive definition for the length of a binary string.

1) According to the recursive definition for the length of a string, what is |001|?

Check Show answer

2) If |x| = 7, then what is |x0|?

Check Show answer

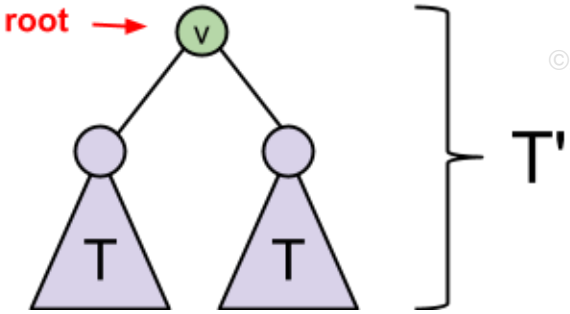
Recursive definition for perfect binary trees.

The next example is a recursive definition for a set of mathematical objects that are not strings. A tree has ***vertices*** (denoted by a circle) and ***edges*** (denoted by line segments) which connect pairs of vertices. Not every collection of vertices and edges is a tree. A formal definition of trees along with their properties is given elsewhere in this material. Here we give a recursive definition for a particular class of trees called perfect binary trees. Each perfect binary tree has a designated vertex called the ***root***.

Definition 7.8.2: Recursive definition for the set of perfect binary trees.

Basis: A single vertex with no edges is a perfect binary tree. The root is the only vertex in the tree.

Recursive rule: If T is a perfect binary tree, then a new perfect binary tree T' can be constructed by taking two copies of T, adding a new vertex v and adding edges between v and the roots of each copy of T. The new vertex v is the root of T'.



PARTICIPATION ACTIVITY 7.8.8: Using the recursive rule to construct perfect binary trees from smaller ones.

Animation captions:

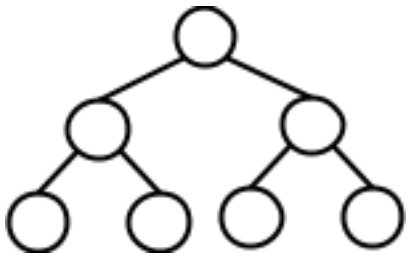
1. The tree T consisting of a single mode is a perfect binary tree.
2. To create a new tree, T is duplicated.
3. A new vertex v is added, along with edges from v to the roots of the two copies of T.
4. The result is a perfect binary tree with three vertices. Call the new tree T.
5. T is duplicated and a new vertex v is added, along with edges from v to the roots of the two copies of T, resulting in a perfect binary tree with 7 vertices.

PARTICIPATION ACTIVITY 7.8.9: How many applications of the recursive rule to build a perfect binary tree?

How many recursive steps need to be applied in order to obtain the following perfect binary trees?

1) 

Check Show answer

2) 

Check Show answer

Additional exercises

Exercise 7.8.1: Properly nested parentheses and curly braces.

About

(a) Give a recursive definition for strings of properly nested parentheses and curly braces. For example, (){}{}() is properly nested but (){} is not properly nested. The empty string should not be included in your definition.

Exercise 7.8.2: Devising recursive definitions for sets of strings.

About

Let A = {a, b}.

(a) Give a recursive definition for A^*.

(b) The set A^+ is the set of strings over the alphabet {a, b} of length at least 1. That is $A^+ = A^* - \{\lambda\}$. Give a recursive definition for A^+.

(c) Let S be the set of all strings from A^* in which there is no b before an a. For example, the strings λ, aa, bbb, and aabbbb all belong to S, but aabab $\notin$ S. Give a recursive definition for the set S. (Hint: a recursive rule can concatenate characters at the beginning or the end of a string.)

(d) For $x \in A^*$, let bCount(x) be the number of occurrences of the character b in x. Give a recursive definition for bCount.

Exercise 7.8.3: Recursive definition for the set of strings over a finite set of characters.

About

Let A be an arbitrary finite set of characters. For example A could be {a, b, c, d, e} or {0, 1, 2}.

(a) Give a recursive definition for A^*.

(b)

Give a recursive definition for |x|, the length of a string $x \in A^*$.

Exercise 7.8.4: Recursive definitions for subsets of binary strings.

About

Give a recursive definition for each subset of the binary strings. A string x should be in the recursively defined set if and only if x has the property described.

(a) The set S consists of all strings with an even number of 1's.

(b) The set S is the set of all binary strings that are palindromes. A string is a palindrome if it is equal to its reverse. For example, 0110 and 11011 are both palindromes.

(c) The set S consists of all strings that have the same number of 0's and 1's.

Exercise 7.8.5: Multiple copies of a string.

About

(a) If x is a string and j is a non-negative integer, then x^j is defined to be j copies of the string concatenated together. For example, if x = 101, then x^4 = 101101101101, and $x^0 = \lambda$. Give a recursive definition for x^j

Exercise 7.8.6: Recursive definitions for subsets of integers.

About

(a) A subset S of the integers is defined recursively as follows:
- Base case: $2 \in S$
- Recursive rule: if $k \in S$, then
  - $k + 5 \in S$
  - $k - 5 \in S$

List the elements of S whose absolute value is less than 20.

(b) A subset T of the integers is defined recursively as follows:
- Base case: $2 \in T$
- Recursive rule: if $k \in T$, then
  - $k + 5 \in T$

List the elements of S whose absolute value is less than 20.

(c) Give a recursive definition for E, the set of all even integers. Recall that 0 is an even

integer. Also an even integer can be negative, such as -2 and -4.

(d) Give a recursive definition for O, the set of all odd integers. Recall that an odd integer can be negative, such as -1 and -3.

Exercise 7.8.7: Interpreting a recursive definition of a set of strings.

About

(a) The recursive definition given below defines a set S of strings over the alphabet {a, b}:

- Base case: λ ∈ S and a ∈ S
- Recursive rule: if x ∈ S then,
  - xb ∈ S
  - xba ∈ S

List all the strings of length at most 3 in S.

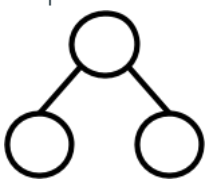
Exercise 7.8.8: The height of a perfect binary tree.

About

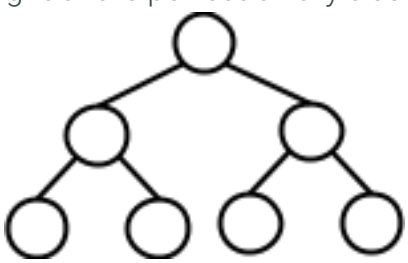
The function h maps every perfect binary tree to a non-negative integer. h(T) is called the height of tree T. The function h is defined recursively as follows:

- If T is the perfect binary tree that consists of a single vertex, then h(T) = 0.
- Suppose that the tree T' is constructed by taking two copies of perfect binary tree T, adding a new vertex v and adding edges between v and the roots of each copy of T. Then h(T') = h(T) + 1.

(a) What is the height of the perfect binary tree shown below?



(b) What is the height of the perfect binary tree shown below?



(c) Describe the function h in words

Exercise 7.8.9: Recursive definition for the number of vertices in a perfect binary tree.

About

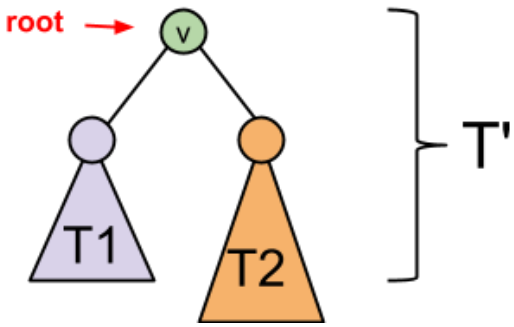
(a) The function v maps every perfect binary tree to a positive integer. v(T) is equal to the number of vertices in T. Give a recursive definition for v(T).

Exercise 7.8.10: A recursive definition for full binary trees.

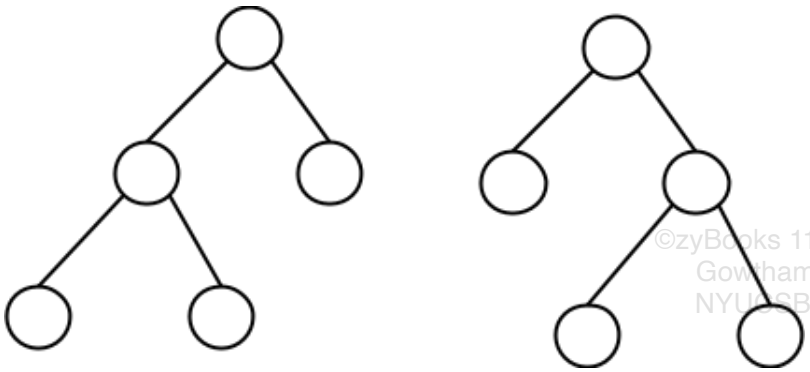
About

Here is a definition for a set of trees called full binary trees.
Basis: A single vertex with no edges is a full binary tree. The root is the only vertex in the tree.

Recursive rule: If T1 and T2 are full binary trees, then a new tree T' can be constructed by first placing T1 to the left of T2, adding a new vertex v at the top and then adding an edge between v and the root of T1 and an edge between v and the root of T2. The new vertex v is the root of T'.



Note that it makes a difference which tree is placed on the left and which tree is placed on the right. For example, the two trees below are considered to be different full binary trees:



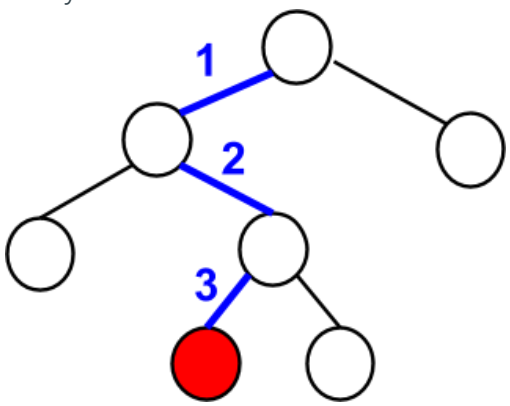
(a) Draw all possible full binary trees with 3 or fewer vertices.

(b) Draw all possible full binary trees with 5 vertices.

(c) Draw all possible full binary trees with 7 vertices.

(d) The function v maps every full binary tree to a positive integer. v(T) is equal to the number of vertices in T. Give a recursive definition for v(T).

(e) The function h maps every full binary tree to a non-negative integer. h(T) is called the height of tree T and is defined to be the maximum number of edges reached in a path that starts at the vertex and always travels downward in the tree. For example, the tree shown below has height 3 because the number of edges reached in a path from the root to the red vertex is 3. Furthermore, there is no path from the root that reaches four edges and always travels downward.



Give a recursive definition for h(T) for every full binary tree T.
(Hint: you may need the max function. For any two numbers, x and y, max(x,y) = x if x ≥ y and max(x, y) = y if y > x.)

7.9 Structural induction

This section has been set as optional by your instructor.

Induction is a powerful technique to prove that all the elements of a recursively defined set have a particular property. ***Structural induction*** is a type of induction used to prove theorems about recursively defined sets that follows the structure of the recursive definition.

As an example, we prove that every properly nested string of left and right parentheses is balanced. A string of parentheses is ***balanced*** if the number of left parentheses is equal to the number of right parentheses. For example, in the string (())(), the number of left parentheses and the number of right parentheses is 3. We will use structural induction to prove that any string of properly nested parentheses is balanced. It will be useful to define some notation to use in the proof. If x is a string of left and right parentheses, then left[x] is equal to the number of left parentheses in x and right[x] is the number of right parentheses in x. The functions "left" and "right" map every string of

parentheses to non-negative integers, regardless of whether the string is balanced or not. Here are some examples to illustrate. The colors highlight left and right parentheses for readability.

- right[ ))()] = 3
- left[ )))] = 0
- left[((()))] = 3

PARTICIPATION ACTIVITY 7.9.1: The left and right functions for strings of parentheses.

Evaluate each of the following:

1) right[())]

Check Show answer

2) left[(())()( ]

Check Show answer

3) If u is a string of parentheses such that left[u] = 3, then what is left[(u)]?

Check Show answer

4) If u and v are strings of parentheses such that right[u] = 2 and right[v] = 5, then what is right[uv]?

Check Show answer

Let P denote the set of properly nested parentheses. The set P is recursively defined below.

Definition 7.9.1: Recursive definition for the set of properly nested parentheses.

- Basis: ( ) ∈ P.
- Recursive rules: If u ∈ P and v ∈ P then:

  1. (u) ∈ P.

2. uv ∈ P.

We will use structural induction to show that if x is a string of properly nested parentheses (x ∈ P), then x has the property of being balanced (left[x] = right[x]). Note that the proof does not imply the converse of the statement: if a string of parentheses has the same number of left and right parentheses then the string is properly nested. In fact the converse of the statement is not true since the string ")(" is balanced but not properly nested.

The animation below shows an outline of the proof along with the elements of a structural induction proof.

PARTICIPATION ACTIVITY 7.9.2: Outline for a proof using structural induction.

Animation content:

Animation captions:

1. Show that for every x in a recursively defined set P, x has a particular property. (Show that every x in P is balanced.)
2. Show that every element in the basis satisfies the property. (Show that "()" is balanced.)
3. Inductive step: The recursive rules show how to construct larger elements in P using smaller elements from P. Assume the property for the smaller elements and prove for the larger ones.
4. Case 1: show that if u is balanced, then (u) is balanced. Case 2: show that if u and v are balanced, then uv is balanced.

PARTICIPATION ACTIVITY 7.9.3: Identifying elements of a proof by structural induction.

The set S is a set of strings defined recursively as follows

- Basis: λ ∈ S
- Recursive rules: if x ∈ S and y ∈ S then,
  - xa ∈ S (Rule 1)
  - xbb ∈ S (Rule 2)
  - xy ∈ S (Rule 3)

Consider a proof that every string in S has an even number of b's. Select whether each proof part belongs in the base case or the inductive step.

1) Assume that x has an even number of b's. Show that xbb also has an even number of b's.
   - ○ Base case

- Inductive step

2) Show that λ has an even number of b's.
   - Base case
   - Inductive step

3) Assume that x and y have an even number of b's. Prove that xy also has an even number of b's.
   - Base case
   - Inductive step

4) Assume that x has an even number of b's. Prove that xa also has an even number of b's.
   - Base case
   - Inductive step

Often the inductive step of an inductive proof is the most difficult part of the proof. The animation below illustrates how the inductive step of the proof that a string of properly nested parentheses is balanced works on some particular examples. The full proof is given afterwards.

PARTICIPATION ACTIVITY 7.9.4: Examples showing the inductive step of a structural induction proof.

Animation content:

undefined

Animation captions:

1. Suppose a string x is obtained by applying Rule 1 to properly nested string u. The string u = (())() has three left parentheses, so left[u] = 3.
2. The string u also has three right parentheses, so right[u] = 3.
3. If x is obtained by applying recursive rule 1 to u, then x = (u) = ((())()).
4. left[x] = left[((())())]. The string ((())()) has one more left parenthesis than (())().
5. Therefore, left[((())())] = 1 + left[(())()] = 1 + 3 = 1 + right[(())()], because left[u] = 3 = right[u].
6. 1 + right[(())()] = right[((())())], because ((())()) has one more right parenthesis than (())().
7. Now consider Rule 2. u = (()). v = (). left[(())] = 2 = right[(())]. Also, left[()] = 1 = right[()].
8. x is obtained by applying Rule 2 to u and v: x = uv = (())().
9. left[x] = left[(())()] = left[(())] + left[()].
10. left[(())] + left[()] = 2 + 1 = right[(())] + right[()].
11. right[(())] + right[()] = right[(())()].

Proof 7.9.1: Properly nested parentheses are balanced.

Theorem: If string x ∈ P, then left[x] = right[x].

Proof.
By induction.

Base case: () ∈ P. left[()] = right[()] = 1.

Inductive step: If x ∈ P, then x was constructed by applying a sequence of recursive rules starting with the string () given in the basis. We consider two cases, depending on the last recursive rule that was applied to construct x.

Case 1: Rule 1 is the last rule applied to construct x. Then x = (u), where u ∈ P. We assume that left[u] = right[u] and prove that left[x] = right[x]. Then $\begin{align} \mbox{left}[x] & = \mbox{left}[~(u)~] &\mbox{because}~x = (u)\\ & = 1 + \mbox{left}[u] &(u)~\mbox{has one more "(" than}~u\\ & = 1 + \mbox{right}[u]~~&\mbox{by the inductive hypothesis}\\ & = \mbox{right}[~(u)~] &(u)~\mbox{has one more ")" than}~u\\ & = \mbox{right}[x] &\mbox{because}~x = (u)\\ \end{align}$

Case 2: rule 2 is the last rule applied to construct x. Then x = uv, where u ∈ P and v ∈ P. We assume that left[u] = right[u] and left[v] = right[v] and then prove that left[x] = right[x]. Then $\begin{align} \mbox{left}[x] & = \mbox{left}[uv]&\mbox{because}~x = uv\\ & = \mbox{left}[u] + \mbox{left}[v]&\\ & = \mbox{right}[u] + \mbox{right}[v]~~&\mbox{by the inductive hypothesis}\\ & = \mbox{right}[uv]~~&\\ & = \mbox{right}[x] &\mbox{because}~x = uv\\ \end{align}$

Therefore, left[x] = right[x]. ■

Structural induction and perfect binary trees

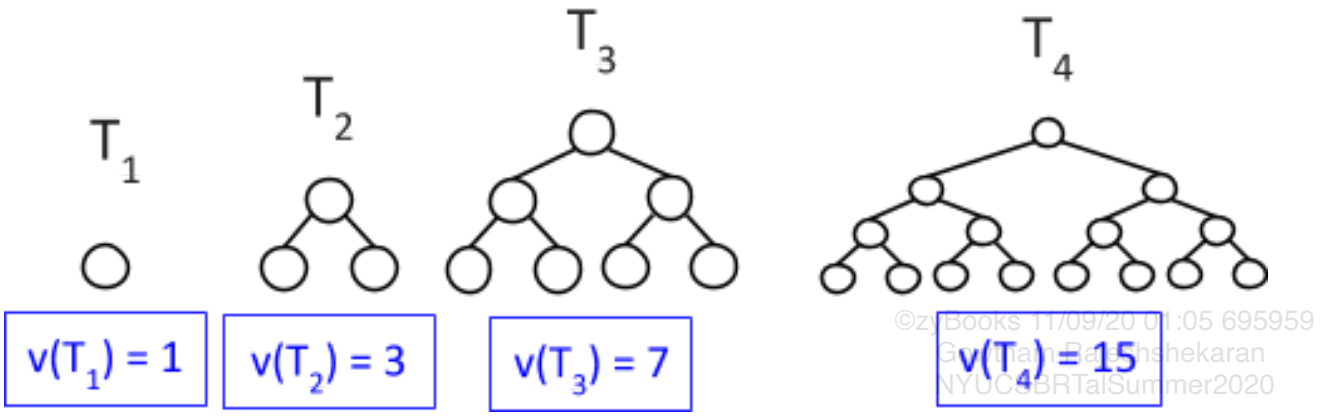
The next example uses structural induction to prove the following theorem for perfect binary trees.

Theorem 7.9.1: Number of vertices in a perfect binary tree: proof using structural induction.

Let T be a perfect binary tree. Then the number of vertices in T is 2^k - 1 for some positive integer k.

If T is a perfect binary tree T, then let v(T) denote the number of vertices in T. The figure below shows a few examples of perfect binary trees and the number of vertices in each.

Figure 7.9.1: Number of vertices in perfect binary trees.



PARTICIPATION ACTIVITY 7.9.5: Using structural induction to prove a theorem about perfect binary trees.

This question outlines a proof of the theorem that the number of vertices in a perfect binary tree is $2^k - 1$ for some positive integer k.

1) The basis in the recursive definition of perfect binary trees says that T_1, the tree with one vertex and no edges, is a perfect binary tree. What fact must be proven in the base case for the proof?
   - $v(T_1) = 1 = 2^1 - 1$.
   - $v(T_1) = 0 = 2^0 - 1$.

2) Let T' be a perfect binary tree created by at least one application of the recursive rule using a smaller perfect binary tree T. Select the statement that corresponds to the inductive hypothesis.
   - $v(T') = 2^k - 1$, for some positive integer k.
   - $v(T') = 2 \cdot v(T) - 1$.
   - $v(T) = 2^k - 1$, for some positive integer k.

3) The recursive rule constructs T' by taking two copies of T, adding a vertex v, and adding edges between v and the roots of each copy of T.

Select which statement is true about the relationship between the number of vertices in T and the number of vertices in T'.

- $v(T) = 2 \cdot v(T') + 1$.
- $v(T') = 2 \cdot v(T) + 1$.
- $v(T') = v(T) + 1$.

4) Here is the argument for the inductive step. In which line is the inductive hypothesis applied?

$$\begin{aligned} v(T') &= 2 \cdot v(T) + 1; && (1) \\ &= 2(2^k - 1) + 1; && (2) \\ &= 2^{k+1} - 1 && (3) \end{aligned}$$

- 1
- 2
- 3

Figure 7.9.2: The proof for the number of vertices in a perfect binary tree.

Proof.
By induction.

Base case: $2^1 - 1 = 1$, so the tree with one vertex has $2^k - 1$ leaves, with $k = 1$.

Inductive step: Let T' be a perfect binary tree that is created with one or more applications of the recursive rule. The last recursive rule that is applied to create T' takes a perfect binary tree T, duplicates T and adds a new vertex v with edges to each of the roots of the two copies of T. We assume that $v(T) = 2^k - 1$, for some positive integer k and prove that $v(T') = 2^j - 1$ for some positive integer j.

The number of vertex in T' is twice the number of vertices in T (because of the two copies of T) plus 1 (because of the vertex v that is added), so $v(T') = 2 \cdot v(T) + 1$. By the inductive hypothesis, $v(T) = 2^k - 1$ for some positive integer k. Therefore

$$v(T') = 2 \cdot v(T) + 1 = 2(2^k - 1) + 1 = 2 \cdot 2^k - 2 + 1 = 2^{k+1} - 1$$

Since k is a positive integer, j = k+1 is also a positive integer. Therefore the number of vertices in T' is $2^j - 1$, where j is a positive integer. ■

Additional exercises

Exercise 7.9.1: Properly nested parentheses and curly braces, cont.

About

(a) Use structural induction to prove that for properly nested parentheses and curly braces:

1. the number of left parentheses is the same as the number of right parentheses, and
2. the number of left curly braces is equal to the number of right curly braces.

Exercise 7.9.2: Proving facts about recursively defined sets of strings.

About

(a) Consider a set of strings defined recursively as follows:

- Base case: a ∈ S
- Recursive rule: if x ∈ S then,
  - xb ∈ S (Rule 1)
  - xa ∈ S (Rule 2)

Prove that every string in S begins with the character a.

(b) Consider a set of strings defined recursively as follows:

- Base case: a ∈ S
- Recursive rules: if x ∈ S then,
  - xb ∈ S (Rule 1)
  - bx ∈ S (Rule 2)

Prove that every string in S contains exactly one a.

(c) Consider a set of strings defined recursively as follows:

- Base case: a ∈ S
- Recursive rules: if x ∈ S then,
  - xb ∈ S (Rule 1)
  - bx ∈ S (Rule 2)
  - axa ∈ S (Rule 3)
  - xaa ∈ S (Rule 4)

Prove that every string in S contains an odd number of a's.
Note that the set S as defined does not contain every string over the alphabet {a, b} that contains an odd number of a's. Therefore, the converse of this statement (if x has an odd number of a's then a ∈ S) is not true. For example, there is no way to create the string "aababa" by applying the recursive rules to the string "a" given in the base case.

(d) Consider a set of strings defined recursively as follows:

- Base case: λ ∈ S
- Recursive rules: if x ∈ S and y ∈ S then,
  - axb ∈ S (Rule 1)
  - bxa ∈ S (Rule 2)

- xy ∈ S (Rule 3)

Prove that every string in S contains the same number of a's and b's.
Note that your proof does not necessarily imply that every string that has the same number of a's and b's is in S.

Exercise 7.9.3: Characterizing the strings in a recursively defined set.

About

The recursive definition given below defines a set S of strings over the alphabet {a, b}:

- Base case: λ ∈ S and a ∈ S
- Recursive rule: if x ∈ S then,
  - xb ∈ S (Rule 1)
  - xba ∈ S (Rule 2)

This problem asks you to prove that the set S is exactly the set of strings over {a, b} which do not contain two or more consecutive a's. In other words, you will prove that x ∈ S if and only if x does not contain two consecutive a's. The two directions of the "if and only if" are proven separately.

(a) Use structural induction to prove that if a string x ∈ S, then x does not have two or more consecutive a's.

(b) Use strong induction on the length of a string x to show that if x does not have two or more consecutive a's, then x ∈ S. Specifically, prove the following statement parameterized by n:
For any n ≥ 0, let x be a string of length n over the alphabet {a, b} that does not have two or more consecutive a's, then x ∈ S.

Exercise 7.9.4: Multiple copies of a string, cont.

About

(a) Use the recursive definition for x^j, where x is a binary string and j is a non-negative integer, to prove that $|x^j| = j \cdot |x|$.
You can use the fact that for any two binary strings, x and y, |xy| = |x| + |y|.

Exercise 7.9.5: Inductive proofs about a subset of the integers.

About

A subset T of the integers is defined recursively as follows:

- Base case: 2 ∈ T
- Recursive rule: if k ∈ T, then
  - k + 5 ∈ T

This problem asks you to prove that T is exactly the set of integers that can be expressed as 5m+2, where m is a non-negative integer. In other words, you will prove that x ∈ T if and only if x = 5m+2, for some non-negative integer m. The two directions of the "if and only if" are proven separately.

(a) Use structural induction to prove that if k ∈ T, then k = 5m + 2, for some non-negative integer m.

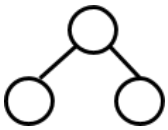
(b) Use structural induction to prove that if k = 5m + 2, for some non-negative integer m, then k ∈ T.

Exercise 7.9.6: Proving facts about perfect binary trees.

About

Here is a recursive definition for a set of trees. The set of trees called TREES.

Base case: The tree shown below is in TREES.



Recursive rule: If T is in TREES, then a new element of TREES called T' can be constructed by taking two copies of T, adding a new vertex v and adding edges between v and the roots of each copy of T. The new vertex v is the root of T'.

Note that the definition of TREES is the same as the definition for perfect binary trees except that the tree consisting of a single vertex is not included in the set TREES.

(a) The degree of a vertex is the number of edges connected to that vertex. Define d(v) to be the degree of vertex v.
Let T be tree in TREES and let r be the root of T. Then
- d(r) = 2
- For every vertex v ≠ r, d(v) = 1 or d(v) = 3.

(b) Let T be a tree in TREES. Let L(T) be the number of vertices in T such that d(v) = 1. Let E(T) be the number of vertices in T such that d(v) > 1. Then L(T) = E(T) + 1.
You can use the fact proven in the previous question.

Exercise 7.9.7: The height of a perfect binary tree, cont.

About

(a) Let T be a perfect binary tree. This problem makes use of the recursive definitions for v(T), the number of vertices in T and h(T) the height of T.
Prove that $h(T) = \log_2(v(T) + 1) - 1$

(Hint: you may use the fact that v(T) is always positive and for any positive number x,

$\log_2 x + 1 = \log_2(2x)$.)

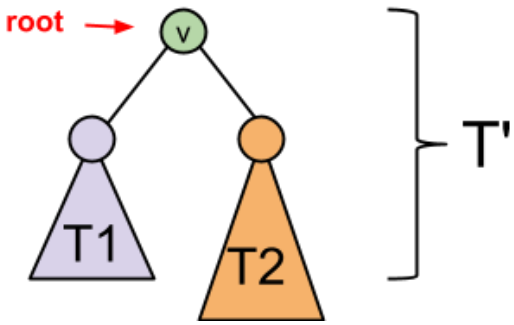
Exercise 7.9.8: A recursive definition for full binary trees, cont.

About

Here is a definition for a set of trees called full binary trees.

Basis: A single vertex with no edges is a full binary tree. The root is the only vertex in the tree.

Recursive rule: If T1 and T2 are full binary trees, then a new tree T' can be constructed by first placing T1 to the left of T2, adding a new vertex v at the time and then adding an edge between v and the root of T1 and an edge between v and the root of T2. The new vertex v is the root of T'.



(a) Prove that a full binary tree has an odd number of vertices. An integer x is odd if x = 2k + 1, for some integer k.

(b) Let T be a full binary tree. This problem makes use of the recursive definitions for v(T), the number of vertices in T and h(T) the height of T.
Prove that $v(T) \geq 2 \cdot h(T) + 1$
(Hint: you may use the fact that h(T) is always non-negative and for any two non-negative numbers, x and y, $x + y \geq \max(x, y)$. This follows from the fact that if $x \geq 0$ and $y \geq 0$, then $x + y \geq x$ and $x + y \geq y$.)

7.10 Recursive algorithms

A ***recursive algorithm*** is an algorithm that calls itself. Like recursively defined sequences and structures, a recursively defined algorithm has a base case in which the output is computed directly on an input of small size or value. On a larger input, the algorithm calls itself on an input of smaller size and uses the result to construct a solution to the larger input. An algorithm's calls to itself are known as ***recursive calls***. Here is an example of a recursive algorithm to compute n!. ***Comments*** in the pseudo-code, which begin with "//", explain the purpose of certain steps and are not part of the algorithm itself.

Example 7.10.1: Recursive algorithm to compute the factorial function.

```
Factorial(n)

Input: A non-negative integer n.
Output: n!

If (n = 0), Return( 1 )
r := Factorial( n - 1 ) // The recursive call

Return( r*n )
```


PARTICIPATION ACTIVITY 7.10.1: An execution of the recursive algorithm to compute 4!.

Animation captions:

1. A call to Factorial with input n = 4 results in a call to Factorial with input n-1 = 3.
2. The call Factorial(3) results in the call Factorial(2), which calls Factorial(1), which calls Factorial(0). When the input n is 0, Factorial returns 1.
3. The call Factorial(0) returns 1 to Factorial(1). Factorial(n) returns n \cdot Factorial(n-1), so Factorial(1) returns 1 \cdot 1 = 1.
4. Factorial(2) returns 2 \cdot 1 = 2. Factorial(3) returns 3 \cdot 2 = 6. The original call to Factorial(4) returns 4 \cdot 6 = 24.

The algorithm given below takes in as input a set A and returns the power set of A. Recall that the power set of A (denoted P(A)) is the set of all subsets of A. The elements of P(A) are subsets of A.

Note that the details of how the set is actually represented and manipulated on a computer are omitted, but there are enough details to run through the algorithm on paper. The base case for the algorithm is when the input A is the empty set. The power set of the empty set is the set containing the empty set as its only element. Then for the recursive step, the algorithm removes an element x from A to obtain A' whose cardinality is strictly smaller than A. The algorithm is then run recursively on A'. All the elements in the power set of A' are also in the power set of A. In addition, the power set of A contains each set in P(A') with the element x added back in.

Example 7.10.2: Recursive algorithm to compute the power set of a set.

```
PowerSet(A)

Input: A set A.
Output: The power set of A.

If (A = ∅), Return( {∅} )

Select an element a ∈ A
A' := A - {a}
P := PowerSet(A')  //the recursive call
P' := P
For each S ∈ P'
   Add S∪{a} to P
End-for

Return( P )
```


PARTICIPATION ACTIVITY 7.10.2: An execution of the recursive algorithm to compute the power set of {a, b, c}.

Animation captions:

1. A call to PowerSet({a, b, c}) removes a to get the set {b,c} and then calls PowerSet({b,c}).
2. A call to PowerSet({b, c}) removes b to get the set {c} and then calls PowerSet({c}) which then calls PowerSet(\varnothing). PowerSet(\varnothing) returns {\varnothing}.
3. When {\varnothing} is returned to PowerSet({c}), the algorithm adds \varnothing \cup{c}= {c} to the power set and returns {\varnothing,{c}}.
4. PowerSet({b,c}) adds b to all the sets and returns {\varnothing,{c},{b},{b,c}}. The PowerSet({a,b,c}) adds a to all the sets and returns {\varnothing,{c},{b},{b,c}{a},{a,c},{a,b},{a,b,c}}

The recursive algorithm below takes as input a string s and outputs the string with all the characters reversed. The algorithm removes the first character from the string, recursively reverses the rest of the string and then adds the first character back at the end of the string.

Example 7.10.3: Recursive algorithm to reverse a string.

```
ReverseString(s)

Input: A string s.
Output: The reverse of s.

If s is the empty string, Return( s )

Let c be the first character in s
Remove c from s
s' := ReverseString(s)
s := string s' with c added to the end

Return( s )
```


PARTICIPATION ACTIVITY 7.10.3: An execution of the recursive algorithm to reverse the string 'POT'.

Animation captions:

1. ReverseString("POT") removes the first char P to get "OT" and calls ReverseString("OT").
2. Then O is removed and ReverseString("T") is called. Then T is removed and ReverseString is called with the empty string which returns the empty string.
3. ReverseString("T") adds T to the end of the empty string to get "T".
4. The string "T" is returned to the call ReverseString("OT") and the O is added to the end of "T" to get "TO".
5. The string "TO" is returned to the call ReverseString("POT") and the P is added to the end of "TO" to get "TOP", which is the final return value.

PARTICIPATION ACTIVITY 7.10.4: A recursive algorithm to compute an exponential.

Here is a recursive algorithm to compute r^n with some lines missing. The input r can be any real number. The input n is assumed to be a non-negative integer.

```
Exponent( r, n )

Input: Real number r and a non-negative integer n.
Output: r^n

--------------------------------  //The base case

p := Exponent( ------ )   //The recursive call

Return( r·p )
```


1) What missing input values should be used in the recursive call?

- (r, n-1)
- (r-1, n-1)
- (r-1, n)

2) What should go in the first blank (labeled "The base case")?

- If (r=0), return(1)
- If (n=1), return(r)
- If (n=0), return(1)

In the above examples, each algorithm has one recursive call. It is also possible for an algorithm to call itself multiple times. Consider the following recursive algorithm to compute Fibonacci numbers:

Example 7.10.4: Recursive algorithm to compute Fibonacci numbers.

```
RecursiveFibonacci(n)

Input: a non-negative integer n.
Output: the Fibonacci number with index n.

If ( n=0 ) Return( 0 )
If ( n=1 ) Return( 1 )

f = RecursiveFibonacci( n-1 ) + RecursiveFibonacci( n-2 )

Return( f )
```

For n = 0 or n = 1, the algorithm returns the correct value without any recursive calls. When n ≥ 2, the call to Fibonacci(n) generates two more recursive calls. The recursive algorithm above is not the most efficient way to compute Fibonacci numbers. As demonstrated in the animation below, the algorithm performs many redundant calculations:

PARTICIPATION ACTIVITY 7.10.5: A recursive algorithm to compute Fibonacci numbers.

Animation captions:

1. RecursiveFibonacci(5) calls RecFib(4) and RecFib(3).
2. RecFib(3) calls RecFib(2) and RecFib(1). RecFib(1) returns from the Base Case.
3. RecFib(2) calls RecFib(1) and RecFib(0) both of which return from the Base Case.
4. The call to RecFib(4) made by the orginal call to RecursiveFibonacci(5) results in eight more recursive calls.
5. RecFib(1) is computed a total of 5 times and RecFib(0) is computed 3 times.

It is much more efficient to compute Fibonacci numbers by using a for-loop and storing the last two numbers in the sequence. The efficiency comes from the fact that the algorithm stores and reuses partial results instead of recomputing.

Example 7.10.5: Non-recursive algorithm to compute Fibonacci numbers.

```
Non-RecursiveFibonacci(n)

Input: A non-negative integer n.
Output: The Fibonacci number with index n.

If ( n = 0 ), Return( 0 )
If ( n = 1 ), Return( 1 )

last = 1
oneBeforeLast = 0

For k = 2 to n
  current := last + oneBeforeLast
  oneBeforeLast := last
  last := current
End-for

Return( current )
```

PARTICIPATION ACTIVITY 7.10.6: Computing Fibonacci numbers.

1) How many addition operations are performed in RecursiveFibonacci(5)?

Check Show answer

2) How many addition operations are performed in Non-RecursiveFibonacci(5)?

Check Show answer

Additional exercises

Exercise 7.10.1: Recursively computing sums of cubes.

About

(a) Give a recursive algorithm to compute the sum of the cubes of the first n positive integers. The input to the algorithm is a positive integer n. The output is \sum_{j=1}^nj^3. The algorithm should be recursive, it should not compute the sum using a closed form expression or a loop.

Exercise 7.10.2: Recursively computing the sum of the first n positive odd integers.

About

(a) Give a recursive algorithm which takes as input a positive integer n and returns the sum of the first n positive odd integers.

Exercise 7.10.3: Recursively computing the maximum and minimum of a sequence.

About

The input to the maximum and minimum problems is a sequence of numbers, $a_1 \ldots a_n$, where n, the length of the sequence, is a positive integer. The function length($a_1 \ldots a_n$)

returns n, the length of the sequence. If n > 1, you can also create a new sequence $a_1 \ldots a_{n-1}$, which is the original sequence $a_1 \ldots a_n$, with the last number a_n omitted.

(a) Give a recursive algorithm which takes as input a sequence of numbers and returns the minimum (i.e., smallest) number in the sequence. Your algorithm should not use a loop.

(b) Give a recursive algorithm which takes as input a sequence of numbers and returns the maximum (i.e., largest) number in the sequence. Your algorithm should not use a loop.

Exercise 7.10.4: Reversing a string and removing blanks.

About

(a) Give a recursive algorithm that takes as input a string s, removes the blank characters and reverses the string. For example, on input "Hello There", the algorithm should return "erehTolleH". The function IsBlank(c) returns a boolean value indicating whether the character c is the blank character. Your algorithm should not use a loop.

Exercise 7.10.5: Product of primes.

About

(a) Give a recursive algorithm that takes as input an integer n which is greater than 1. The algorithm should return the product of all the primes in the range 2 through n. For example, on input 7, the algorithm will return 210, which the product of 2, 3, 5, and 7. The function IsPrime(x) takes a positive integer x and returns a boolean value indicating whether x is prime.

Exercise 7.10.6: Recursively computing the product of two non-negative integers.

About

(a) Give a recursive algorithm that takes as input two non-negative integers x and y and returns the product of x and y. The only arithmetic operations your algorithm can perform are addition or subtraction. Your algorithm should have no loops.

Exercise 7.10.7: Recursively computing the sum of two non-negative integers.

About

(a) Give a recursive algorithm that takes as input two non-negative integers x and y and returns the sum of x and y. The only arithmetic operations your algorithm can perform are Increment(x) which returns x+1 and Decrement(x) which returns x-1

perform are increment(x) which returns x+1 and decrement(x) which returns x-1. Your algorithm should have no loops.

Exercise 7.10.8: Recursively computing the set of all binary strings of a fixed length.

About

(a) Give a recursive algorithm that takes as input a non-negative integer n and returns a set containing all binary strings of length n. Here are the operations on strings and sets you can use:

- Initialize an empty set S.
- Add a string x to S.
- y := 0x. This operation adds a 0 to the beginning of string x and assigns the result to string y.
- y := 1x. This operation adds a 1 to the beginning of string x and assigns the result to string y.
- return(S)
- A looping structure that performs an operation on every string in a set S:

```
for every x in S,
    // perform some sequence of steps with string x.
```


Exercise 7.10.9: Recursively computing a number raised to an exponent that is a power of 2.

About

(a) Give a recursive algorithm whose input is a real number r and a non-negative integer n, and whose output is $r^{(2^n)}$. **Note that the exponent of r is 2^n.** Your algorithm should only use addition and multiplication operations and should not contain any loops.

7.11 Induction and recursive algorithms

Induction is a useful proof technique to prove facts about recursive algorithms such as establishing that a recursive algorithm gives the correct output for every possible valid input. The first step is to prove that the base case works correctly. The inductive step then shows that the algorithm returns the correct value assuming that all of the recursive calls return the correct values.

An inductive proof can be used to show that the recursive algorithm ReverseString returns the reverse of the input string.

PARTICIPATION ACTIVITY 7.11.1: Overview of the proof of correctness for ReverseString.

Animation captions:

1. On a call to ReverseString(s), if s is empty (the base case), then the reverse of s is s.
2. The characters in s are numbered $c_1c_2c_3\ldots c_kc_{k+1}$. When the first character is removed from s, the result is $c_2c_3\ldots c_kc_{k+1}$.
3. By the inductive hypothesis, the recursive call returns the correct output, s' = $c_{k+1}c_k\ldots c_3c_2$.
4. The algorithm returns s' with c_1 added to the end, which is $c_{k+1}c_k\ldots c_3c_2c_1$, the correct reverse of s.

Theorems proven by induction are usually parameterized by an index, such as "For all $n \geq 0$, P(n) is true". The inductive step assumes that P(k) is true and then proves that P(k+1) is true. The most natural statement of the fact that ReverseString works correctly does not have an explicit parameter n:

For every input string s, ReverseString returns the reverse of string s.

Therefore, the proof starts out with the statement: "By induction on the length of the string", which means that the length of the string will be the parameter n. The theorem being proven is implicitly:

If s is a string of length n, then ReverseString(s) returns the reverse of string s.

Figure 7.11.1: Proof of correctness for ReverseString.

Proof.

- By induction on the length of the string.
- **Base case:** s is a string of length 0. Since s has no characters, the reverse of s is s, which is the string returned by the algorithm.
- **Inductive step:** Assume that for $k \geq 0$, ReverseString returns the correct output for any string of length k. Then we will prove that ReverseString returns the correct output for any string of length k+1.
    - Let s be a string of length k + 1. The characters in s are numbered as $s = c_1c_2\ldots c_kc_{k+1}$.
    - The algorithm ReverseString makes a recursive call whose input is the string s with the first character removed: $c_2\ldots c_kc_{k+1}$.
    - The string $c_2\ldots c_kc_{k+1}$ has k characters, so by the inductive hypothesis, the recursive call to ReverseString correctly returns the reverse of string $c_2\ldots c_kc_{k+1}$, which is $c_{k+1}c_k\ldots c_2$.
    - The algorithm ReverseString on input s, returns $c_{k+1}c_k\ldots c_2$ with c_1 added to the end: $c_{k+1}c_k\ldots c_2c_1$, which is in fact the original string s with the characters reversed. ■

An inductive proof is also used to show that the recursive algorithm PowerSet works correctly. The input is a finite set A and the output should be the power set of A, the set whose elements are all the subsets of A. The proof of correctness must show that if PowerSet(A) returns P, then P contains all of the subsets of A. In addition, the proof must establish that P does not have any additional elements that are are not subsets of A. In other words, the proof must show that $X \subseteq A$ if and only if $X \in P$.

Figure 7.11.2: Recursive algorithm to compute the the power set of a set.


```
PowerSet(A)

Input: Finite set A.
Output: The power set of A.

1. If A = ∅, Return({∅})

2. Select an element a ∈ A
3. A' := A - {a}
4. P := PowerSet(A')  // The recursive call
5. For each S ∈ P
6.     Add S ∪ {a} to P

7. Return( P )
```

PARTICIPATION ACTIVITY 7.11.2: Proof of correctness for PowerSet.

Theorem: For any finite set A, PowerSet(A) returns the correct power set of A.

1) The first statement of the proof is "By induction on the size of the input set A." Select the phrase to fill in the blank to get a restatement of the theorem parameterized by n:

   Let A be _____. The PowerSet(A) returns the correct power set of A.

   - The empty set.
   - A finite set.
   - A finite set of size n.

2) The base case is proven for n = 0.

What is the set A, when n = 0?

- the empty set
- an arbitrary set
- an arbitrary finite set

3) The inductive hypothesis assumes that for every k ≥ 0, the theorem holds for any set with k elements. What must be proven in the inductive step?

- If A has 0 elements then PowerSet(A) returns the correct power set of A.
- If A has k elements then PowerSet(A) returns the correct power set of A.
- If A has k + 1 elements then PowerSet(A) returns the correct power set of A.

4) Suppose that a set A has k + 1 elements and a ∈ A. Then what is the size of the set A - {a}?

- k - 1
- k
- k + 1

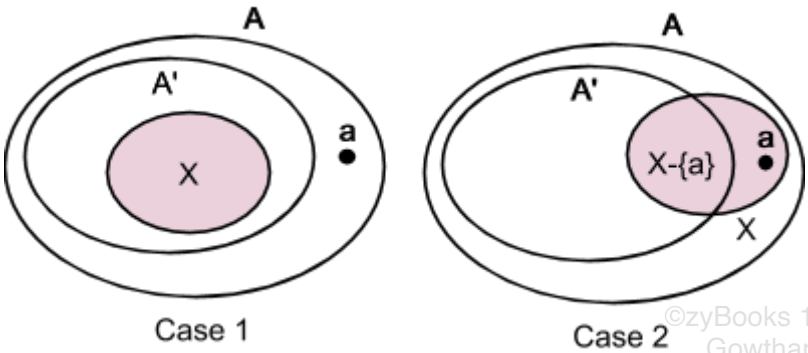
Figure 7.11.3: Proof of correctness for PowerSet.

Proof.

By induction on the size of the input set A.

Base case: If |A| = 0, then A = ∅. On input A = ∅, Power(A) returns {∅} which is exactly the power set of A.

Inductive step: Assume that PowerSet works correctly on an input set of size k and prove that PowerSet works correctly on an input set of size k + 1. Let A be a set with k + 1 elements. The algorithm PowerSet(A) removes an element a from the set A to obtain a new set A' with one less element, so A' = A - {a}, and |A'| = k + 1 - 1 = k. Let P' be the set returned by PowerSet(A') and P the set returned by PowerSet(A). The inductive hypothesis says that P' is exactly the power set of A'. We will use the inductive hypothesis to show that P is the power set of A. That is X ⊆ A if and only if X ∈ P. The first direction is to prove that if X ⊆ A, then X ∈ P. There are two cases pictured below:



Case 1: a ∉ X. Then X is also a subset of A'. Since X is in the power set of A', by the inductive hypothesis, X ∈ P'. The algorithm only adds more subsets to P' to get P, so X is also in P.

Case 2: a ∈ X. Then X - {a} is a subset of A - {a} = A'. By the inductive hypothesis, X - {a} is included in P'. Therefore, in Step 6, the algorithm includes X - {a} ∪ {a} = X in P.

Therefore, if X ⊆ A, then X ∈ P.

The final step is to show that if X ∈ P, then X ⊆ A. Any set added to P is either in P', and therefore a subset of of A', or is S ∪ {a}, where S is a subset of A'. Therefore if a set is added to P, then the set is a subset of A.

■

FastExpRec is a recursive algorithm that computes r^n, where r can be any real number and n is a non-negative integer. FastExpRec is much faster than the algorithm that just multiplies r times itself n - 1 times. However, the fact that FastExpRec works is much less intuitive. An inductive proof is used to show FastExpRec works correctly.

The correctness of the algorithm FastExpRec, hinges on the fact that if n is even then n = 2x for some integer x, and if n is odd, then n = 2x + 1, for some integer x. In either case $\lfloor n/2 \rfloor = x$: $$\begin{aligned} \left\lfloor \frac{2x}{2} \right\rfloor &= \left\lfloor x \right\rfloor = x \\ &\\ \left\lfloor \frac{2x+1}{2} \right\rfloor &= \left\lfloor x + \frac 1 2 \right\rfloor = x \\ \end{aligned}$$

PARTICIPATION ACTIVITY 7.11.3: Overview of the proof of correctness for FastExpRec.

Animation captions:

1. On a call to FastExpRec(r,n), the smallest value for n is 0 because n must be non-negative. FastExpRec(r,0) returns r^0 = 1.
2. x = \lfloor n/2 \rfloor. If n is even, n = 2x. If n is odd, n = 2x+1.
3. By the inductive hypothesis, FastExpRec(r,x) returns r^x.

4. If n is even, the algorithm returns the square of FastExpRec(r,x), which is (r^x)^2 = r^{2x} = r^n.
5. If n is odd, the algorithm returns the square of FastExpRec(r,x) times r, which is (r^x)^2r = r^{2x}r = r^{2x+1} = r^n.

Figure 7.11.4: Proof of correctness for FastExpRec.

Proof.
By induction on n.

Base case: n = 0. When n = 0, $r^n = 1$. The algorithm returns 1.

Inductive Step: We show that if FastExpRec(r, x) returns r^x, for any x in the range from 0 through k, then FastExpRec(r, k + 1) returns r^{k+1}. There are two cases, depending on whether k + 1 is odd or even.

Case 1: k + 1 is even. Then k + 1 = 2x, for some integer x and \lfloor (k + 1)/2 \rfloor = x. The integer x is non-negative and less than k + 1, so x falls in the range from 0 through k. By the inductive hypothesis, the recursive call to FastExpRec(r, x) returns r^x. The call to FastExpRec(k + 1) returns y^2 = (\mbox{FastExpRec}(r, x))^2 = (r^x)^2 = r^{2x} = r^{k+1}

Case 2: k + 1 is odd. Then k + 1 = 2x + 1, for some integer x and \lfloor (k + 1)/2 \rfloor = x. The integer x is non-negative and less than k + 1, so x falls in the range from 0 through k. By the inductive hypothesis, the recursive call to FastExpRec(r, x) returns r^x. The call to FastExpRec(k + 1) returns y^2\cdot r = (\mbox{FastExpRec}(r, x))^2\cdot r = (r^x)^2\cdot r = r^{2x}\cdot r = r^{2x + 1} = r^{k + 1}

Therefore, the call to FastExpRec(r, k + 1) returns r^{k+1}. ■

PARTICIPATION ACTIVITY 7.11.4: Proof of correctness of Exponent.

The algorithm Exponent is a recursive algorithm which computes r^n in a different way than FastExpRec. The input r can be any real number. The input n is assumed to be a non-negative integer.

```
Exponent( r, n )

Input: Real number r and a non-negative integer n.
Output: r^n

If ( n = 0 ), Return( 1 )  //The base case

p := Exponent(r, n-1)

Return( r·p )
```


1) The correctness of algorithm Exponent(r, n) is proven by induction on n. What would be proven in the base case?
   - Exponent(0, n) returns 0
   - Exponent(r, 0) returns 1
   - Exponent(r, 1) returns r

2) Suppose that the inductive hypothesis is that Exponent(r, k) returns r^k. What fact must be proven in the inductive step?
   - Exponent(r, k+1) returns r^{k+1}.
   - Exponent(r+1, k) returns $(r+1)^k$.
   - Exponent(r, k-1) returns r^{k-1}.

3) The inductive step shows that the value returned by Exponent(r, k+1) is:

   $p \cdot r = \text{Exponent}(r, k) \cdot r = r^k \cdot r = r^{k+1}$

   Which equality makes use of the inductive hypothesis?
   - $p \cdot r = \text{Exponent}(r, k) \cdot r$
   - $\text{Exponent}(r, k) \cdot r = r^k \cdot r$
   - $r^k \cdot r = r^{k+1}$

Additional exercises

Exercise 7.11.1: Recursively computing sums of cubes, cont.

About

(a) Use induction to prove that your algorithm to compute the sum of the cubes of the first n positive integers returns the correct value for every positive integer input.

Exercise 7.11.2: Recursively computing the sum of the first n positive odd integers, cont.

About

(a) Use induction to prove that your algorithm to compute the sum of the first n positive odd integers returns the correct value for every positive integer input.

Exercise 7.11.3: Recursively computing the maximum and minimum of a sequence, cont.

About

(a) Use induction to prove that your algorithm to compute the minimum of a sequence outputs the correct value for every non-empty sequence of numbers.
(Hint: the minimum value of a sequence $a_1 \ldots a_n$ is the unique value x such that $x = a_j$ for some $1 \leq j \leq n$ and $x \leq a_i$ for every $1 \leq i \leq n$.)

(b) Use induction to prove that your algorithm to compute the maximum of a sequence outputs the correct value for every non-empty sequence of numbers.
(Hint: the maximum value of a sequence $a_1 \ldots a_n$ is the unique value x such that $x = a_j$ for some $1 \leq j \leq n$ and $x \geq a_i$ for every $1 \leq i \leq n$.)

Exercise 7.11.4: Reversing a string and removing blanks, cont.

About

(a) Use induction to prove that your algorithm to reverse a string and remove blank characters outputs the correct string for every possible input string.

Exercise 7.11.5: Product of primes, cont.

About

(a) Use induction to prove that your algorithm to compute the product of primes in the range 2 through n computes the correct value for every integer n which is strictly

greater than one.

Exercise 7.11.6: Recursively computing the product of two non-negative integers, cont.

About

(a) Use induction to prove that your algorithm that recursively computes the product of x and y returns the correct output for every pair (x, y) where x and y are non-negative integers.

Exercise 7.11.7: Recursively computing the sum of two non-negative integers, cont.

About

(a) Use induction to prove that your algorithm that recursively computes the sum of x and y returns the correct output for every pair (x, y) where x and y are non-negative integers.

Exercise 7.11.8: Recursively computing the set of all binary strings of a fixed length, cont.

About

(a) Use induction to prove that your algorithm to compute the set of all binary strings of length n returns the correct set for every input n, where n is a non-negative integer.

Exercise 7.11.9: Recursively computing a number raised to an exponent that is a power of 2, cont.

About

(a) Use induction to prove that your algorithm to compute $r^{(2^n)}$ returns the correct value for every input pair (r, n), where r is a real number and n is a non-negative integer.

Exercise 7.11.10: A fast algorithm to multiply two non-negative integers.

About

(a)

The following algorithm takes as input two non-negative integers, x and y, and returns the product of x and y.

```
FastMult(x, y)

Input: Two non-negative integers, x and y
Output: The product of x and y

If (y = 0), Return( 0 )

z := \lfloor y/2 \rfloor
p := FastMult(x, z) // The recursive call

If (y is even)
    Return( 2 · p )
Else
    Return( 2 · p + x )
End-if
```


Use induction to prove that the algorithm returns the correct value.

7.12 Analyzing the time complexity of recursive algorithms

Deriving the recurrence relation

The ***time complexity*** of an algorithm is a function T(n) indicating the number of atomic operations performed by the algorithm on an input of size n. For a recursive algorithm, the function T(n) is defined recursively by a recurrence relation and initial values. For example, suppose that the function T(n) indicates the number of atomic operations performed by the factorial algorithm given below on input n.

```
Factorial(n)

If (n = 0), Return(1)

r := Factorial(n-1)

Return( r*n )
```

- Initial value: The initial value for the function T is the number of operations performed during the base case. The base case occurs for the factorial algorithm when n = 0, so the initial value would be the value for T(0).
- Recurrence relation: For n \ge 1, the value of T(n) is defined by a recurrence relation that includes the running time for the recursive calls plus the additional operations performed outside the recursive calls.

PARTICIPATION ACTIVITY | 7.12.1: Deriving the recurrence relation that defines the time complexity of a recursive algorithm.

Animation captions:

1. T(n) = number of atomic operations performed by the factorial algorithm on input n.
2. If n = 0, two atomic operations are performed: one to test n = 0 and one to return the value of 1. Therefore T(0) = 2.
3. The number of operations performed in the recursive call to Factorial on input n-1 is T(n-1).
4. There are 5 additional operations, so the recurrence relation is T(n) = T(n-1) + 5.

PARTICIPATION ACTIVITY | 7.12.2: Deriving a recurrence relation to determine the asymptotic complexity of a recursive algorithm.

The function PowerRecursive below takes inputs r and n, where r is any real number and n is a non-negative integer. The output is r^n.

```
PowerRecursive(r, n)

If (n = 0), Return (1)

y := PowerRecursive(r, n-1)

Return (r\cdot y)
```

The number of atomic operations performed by PowerRecursive does not depend on the value of r. The function T(n) is the number of atomic operations performed by the PowerRecursive on input n. The questions below derive the recurrence relation for the time complexity of the algorithm PowerRecursive.

1) What is the number of atomic operations performed by the recursive call in PowerRecursive?
   - ○ T(n)
   - ○ T(n-1)
   - ○ T(r-1)

2) How many atomic operations are

performed by PowerRecursive outside of the recursive call if the exponent n is greater than 0?

- 1
- 5
- n-1

3) What is the correct recurrence relation for T(n)?

- T(n) = T(n-1)
- T(n) = n + 5
- T(n) = T(n-1) + 5

Overview of analyzing recursive algorithms: A two step process

The animation above shows that the time complexity of the factorial function T(n) is described by the initial value and recurrence relation:

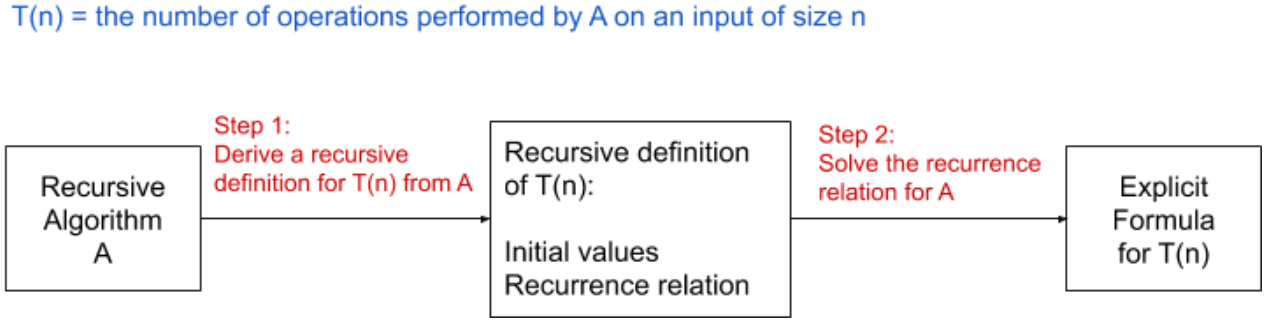
- T(0) = 2
- T(n) = T(n-1) + 5, for n \ge 1

The recursive definition for T is not useful in determining the actual number of operations performed on a particular input. For example, the recursive definition does not provide a convenient way to determine T(100), the number of operations performed by the algorithm on an input of size 100.

An explicit formula for T(n), that does not depend on the value of T for smaller inputs, is more useful in understanding the time complexity of the algorithm. It turns out that the explicit formula for the factorial algorithm is T(n) = 5n + 2.

After deriving the recurrence relation for T(n) from a recursive algorithm, the next step is to solve the recurrence relation by finding an explicit formula for T(n). This section focuses on deriving a recurrence relation from a recursive algorithm. Solving recurrence relations is covered elsewhere in this material.

Figure 7.12.1: The two steps in analyzing the time complexity of a recursive algorithm.



PARTICIPATION ACTIVITY 7.12.3: Recursively defined functions and explicit formulas.

Identify whether the definition of the function T is a recursive definition or an explicit formula.

1) T(n) = 3n -2
   - Recursive definition
   - Explicit formula

2) T(n) = 3 \cdot T(\lceil n/3 \rceil) + n^2 + 2, for n \ge 2 and T(1) = 3
   - Recursive definition
   - Explicit formula

3) T(n) = 3^n + n^2 + 7
   - Recursive definition
   - Explicit formula

Simplifying recurrence relations for asymptotic analysis

The exact number of operations performed by an algorithm is less important than the asymptotic complexity of the algorithm, described using Θ notation. For example, it is more important to know that the time complexity of the factorial function is Θ(n), rather than 5n + 2.

Some of the constants in a recurrence relation do not affect the asymptotic growth of the final explicit formula. For example, the recurrences T(n) = T(n-1) + 5n and T(n) = T(n-1) + 7n both result in explicit formulas that are \Theta(n^2). Therefore, both recurrence relations can be simplified as T(n) = T(n-1) + Θ(n).

The initial value for T, such as the value of T(0), is omitted in determining the asymptotic time complexity of a recursive algorithm because the number of operations performed in the base case is almost always Θ(1). The exact constant does not affect that asymptotic complexity of the algorithm.

PARTICIPATION ACTIVITY 7.12.4: Rules for simplifying recurrence relations for asymptotic analysis.

Animation captions:

1. Additive functions of n can be replaced with \Theta notation.
2. For example, any recurrence of the form T(n) = T(n-1) + \Theta(n^2) will have explicit formula \Theta(n^3).

3. Terms such as T(\lceil n/c \rceil) and T(\lfloor n/c \rfloor) can be replaced with T(n/c) for any constant c.
4. Recurrences T(n) = 2 \cdot T(\lceil n/2 \rceil) + \Theta(n) and T(n) = 2 \cdot T(\lfloor n/2 \rfloor) + \Theta(n) can be simplified as T(n) = 2 \cdot T(n/2) + \Theta(n).
5. Constant factors in front of T(·) are important in the asymptotic growth of T(n) and can not be changed.
6. The two recurrences T(n) = 4 \cdot T(n/2) + \Theta(n) and T(n) = 2 \cdot T(n/2) + \Theta(n) result in explicit. formulas with different asymptotic growth.

PARTICIPATION ACTIVITY 7.12.5: Simplifying recurrence relations in determining the asymptotic growth of functions.

Indicate whether one recurrence relation can replace the other in determining the asymptotic growth of T(n).

1) T(n) = 3 \cdot T(n/2) + 7n
T(n) = T(n/2) + \Theta(n)

- Yes
- No

2) T(n) = T(\lfloor n/2 \rfloor) + 5 n^2 + 10
T(n) = T(n/2) + \Theta(n^2)

- Yes
- No

3) T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)
T(n) = 2 \cdot T(n/2) + \Theta(n)

- Yes
- No

4) T(n) = T(n-1) + 3 n^2 + 8n
T(n) = T(n-1) + \Theta(n^3)

- Yes
- No

Deriving a simplified recurrence relation for asymptotic analysis

The specific constants that arise in counting each atomic operation in a recursive algorithm do not affect the algorithm's asymptotic complexity. The animation below shows how the derivation of a recurrence relation can be simplified if the goal is to determine the algorithm's asymptotic complexity instead of determining the exact number of operations performed.

PARTICIPATION ACTIVITY 7.12.6: Deriving the recurrence relation that defines the asymptotic time complexity of a recursive algorithm.

Animation captions:

1. T(n) = number of atomic operations performed by the factorial algorithm on input n.
2. The number of operations performed in the recursive call to Factorial on input n-1 is T(n-1).
3. There are \Theta(1) additional operations, so the recurrence relation is T(n) = T(n-1) + \Theta(1).

PARTICIPATION ACTIVITY 7.12.7: Deriving a recurrence relation to determine the asymptotic complexity of a recursive algorithm.

The function FastPowerRecursive below takes inputs r and n, where r is any real number and n is a non-negative integer. The output is r^n.

```
FastPowerRecursive(r, n)

If (n = 0), Return (1)

x := \lfloor n/2 \rfloor
y := FastPowerRecursive(r, x)

If (n is even), Return (y\cdot y)
If (n is odd), Return (y\cdot y \cdot r)
```

The number of atomic operations performed by FastPowerRecursive does not depend on the value of r. The function T(n) is the number of atomic operations performed by the FastPowerRecursive on input n. The questions below derive the recurrence relation for the asymptotic complexity of the algorithm FastPowerRecursive.

1) What is the correct T(\cdot) term in the recurrence relation for T(n)?
   - T(n)
   - T(n-1)
   - T(n/2)

2) How many atomic operations are performed by FastPowerRecursive outside of the recursive call? You can assume that the test to determine whether n is even or odd requires Θ(1) operations.
   - \Theta(1)
   -

\Theta(r)

- \Theta(n)

3) What is the correct recurrence relation for T(n)?

- T(n) = T(n/2)
- T(n) = T(n/2) + \Theta(1)
- T(n) = 2 \cdot T(n/2) + \Theta(1)

Additional exercises

Exercise 7.12.1: Simplifying recurrence relations for asymptotic analysis.

About

Simplify each recurrence relation as much as possible. The explicit formula for the function should have the same asymptotic growth as the original recurrence relation.

(a)

T(n) = T(n-1) + 5n^3 + 4n

(b)

T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) +7n

(c)

T(n) = 3 \cdot T(\lfloor n/2 \rfloor) + 14

(d)

T(n) = 3 \cdot T(\lceil n/3 \rceil) + 4n + 6 n \log n

Exercise 7.12.2: Recursively computing sums of cubes, cont.

About

(a) Give the recurrence relation to describe the asymptotic time complexity of your algorithm to compute the sum of the cubes of the first n positive integers.

Exercise 7.12.3: Recursively computing the product of two non-negative integers, cont.

About

(a) Give the recurrence relation to describe the asymptotic time complexity of your algorithm that recursively computes the product of x and y, where x and y are non-negative integers.

Exercise 7.12.4: Recursively computing a number raised to an exponent that is a power of 2, cont.

About

(a) Give the recurrence relation to describe the asymptotic time complexity of your algorithm to compute r^{2^n}.

Exercise 7.12.5: Recurrence relation for RecursiveFibonacci.

About

(a) The algorithm given below is a recursive algorithm to compute the n^{th} Fibonacci number. Give a recurrence relation to describe the asymptotic complexity of the algorithm as a function of n, the value of the input integer.

```
RecursiveFibonacci(n)

If (n = 0), Return( 1 )
If (n = 1), Return( 1 )

f := RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2)

Return( f )
```

Exercise 7.12.6: A fast algorithm to multiply two non-negative integers, cont.

About

(a)

The following algorithm takes as input two non-negative integers, x and y, and returns the product of x and y.

```
FastMult(x, y)

Input: Two non-negative integers, x and y
Output: The product of x and y

If (y = 0), Return( 0 )

z := \lfloor y/2 \rfloor
p := FastMult(x, z) // The recursive call

If (y is even)
    Return( 2 · p )
Else
    Return( 2 · p + x )
End-if
```


Give a recurrence relation to describe the asymptotic complexity of the algorithm.

7.13 Divide-and-conquer algorithms: Introduction and mergesort

This section has been set as optional by your instructor.

Divide-and-conquer algorithms

A ***divide-and-conquer*** algorithm solves a problem recursively by breaking the original input into smaller subproblems of roughly equal size. There are three basic steps in a divide-and-conquer algorithm:

- Break the input into smaller subproblems of the same type on smaller inputs
- Solve each subproblem recursively
- Combine the solutions of the subproblems to obtain a solution to the original problem

For example, a divide-and-conquer algorithm to find the smallest number in a list would break the list into two halves, recursively find the smallest number in each half, compare the two smallest numbers from each half, and return the smaller of the two. The base case occurs when the list only has one item, in which case the smallest number is the only number in the list.

```
FindMin(L)

If L has only one item x, Return(x)

Break list L into two lists, A and B

a := FindMin(A)
b := FindMin(B)

If ( a \le b ), then Return(a)
Else Return(b)
```


PARTICIPATION ACTIVITY 7.13.1: Illustration of divide-and-conquer FindMin.

Animation captions:

1. Divide the input list (17, 3, 9, 11, 21, 4, 7, 2) into two lists: (17, 3, 9, 11) and (21, 4, 7, 2).
2. Recursively find the smallest item in the first list (17, 3, 9, 11). The base case occurs when the list has only one item (17). FindMin returns 17.
3. When the two recursive calls are complete, FindMin returns the smaller of the two values.
4. The recursive call to (17, 3, 9, 11) returns 3.
5. Now FindMin makes a recursive call on the second half of the list (21, 4, 7, 2) which returns 2.
6. The recursive call on (17, 3, 9, 11) returns 3. The recursive call on (21, 4, 7, 2) returns 2. FindMin returns the smaller of 2 and 3, which is 2.

PARTICIPATION ACTIVITY 7.13.2: Divide-and-conquer compute sum.

The instructions below give the basic steps in a divide-and-conquer algorithm that computes the sum of all the numbers in a list:

- Break the list into two lists, each with half the numbers: List1 and List2
- R1 := the value returned from a recursive call to the algorithm with input List1.
- R2 := the value returned from a recursive call to the algorithm with input List2.
- Return (?)

1) What value should the algorithm return?

- 2 \cdot R1
- R2
- R1 + R2
- min{R1, R2}

2) Suppose the base case occurs when there is only one number in the list (x). What value should the algorithm return for the base case?

- 0
- 1
- x

Mergesort

A ***sorting algorithm*** takes as input a list of items and returns the same list sorted in ascending order. On input (12, 4, 7, 9), for example, the correct output is (4, 7, 9, 12). The items to be sorted must have a well-defined ordering. For example, integers can be sorted by value and names can be sorted according to alphabetical order.

Mergesort is a divide-and-conquer algorithm that sorts a list of items recursively by dividing the list into two sub-lists of roughly half the size, recursively sorting each sub-list and merging the sorted sub-lists. Mergesort would sort the list (17, 3, 9, 11, 21, 4, 7, 1) in four basic steps:

Break the list (17, 3, 9, 11, 21, 4, 7, 1) into two sub-lists: (17, 3, 9, 11) and (21, 4, 7, 1).

Make a recursive call to mergesort with input (17, 3, 9, 11), which returns (3, 9, 11, 17).

Make a recursive call to mergesort with input (21, 4, 7, 1), which returns (1, 4, 7, 21).

Merge the sorted sub-lists to obtain the final sorted list (1, 3, 4, 7, 9, 11, 17, 21).

The base case occurs when the input list only has one item, in which case the list is already sorted.

PARTICIPATION ACTIVITY 7.13.3: Recursion tree for mergesort.

Animation captions:

1. On input (17, 2, 9, 11, 21, 4, 7, 1), recursively call mergesort on (17, 2, 9, 11). Continue to call mergesort on left half of the list until base case with input (17) is reached.
2. Return (17) as the list is already sorted. In each call to mergesort, when both sub-lists have been sorted, merge the two sub-lists.
3. The list (2, 9, 11, 17) is returned from the recursive call to (17, 2, 9, 11). Next make a recursive call on the right half of the input list: (21, 4, 7, 1).
4. The recursive call on the right half of the list returns (1, 4, 7, 12). Now that both halves have been sorted, merge the two sorted sub-lists to get (1, 2, 4, 7, 9, 11, 17, 21), the final sorted list which is returned.

PARTICIPATION ACTIVITY 7.13.4: Recursive calls in mergesort.

1) If mergesort is called an input (5, 4, 3,

2, 9, 8, 7, 6), then what would be the output of the two recursive calls?

- (5, 4, 3, 2) and (9, 8, 7, 6)
- (2, 3, 4, 5) and (6, 7, 8, 9)
- (2, 3, 4, 5, 6, 7, 8, 9)

The queue data structure

A ***data structure*** is a way of organizing data in a computer so that the data can be accessed and updated efficiently. A programmer selects a data structure for an application depending on what particular operations she would like to perform on the data. The version of MergeSort presented in this material uses a data structure called a queue. A ***queue*** maintains items in an ordered lists. New items can be added to one end of the queue called the ***back***. Items can be removed from the other end of the queue called the ***front***. A queue operates like a line at the grocery store in which the customers are items. Customers join the line at the back and leave the line from the front. The details of how to implement a queue are beyond the scope of this material. The important fact for analyzing MergeSort is that the following operations can be implemented in $\Theta(1)$ time:

- A := createEmptyQueue(). Creates a queue with no items and assigns the queue to variable A.
- x := front(A). Returns the item at the front of queue A but does not change A.
- add(x, A). Adds item x to the back of queue A.
- x := remove(A). Removes the item x at the front of queue A and returns x.
- n := size(A). Returns a non-negative integer whose value is the number of items in queue A.

The operations front(A) and remove(A) result in an error if A is empty.

The version of MergeSort presented in this material sorts by moving items between different queues. It is more efficient to implement MergeSort by rearranging the items within a single list or array. However, using queues makes the pseudo-code simpler. All versions of MergeSort run in time $\Theta(n \log n)$ on a list of n items, so the efficiency of different versions only differ by constant factors.

PARTICIPATION ACTIVITY 7.13.5: Simulating operations on a queue.

Animation captions:

1. The operations create an empty list called A and add three items, 6, 7, and 5, to A. The list A is now (6, 7, 5).
2. The operation front(A) returns 6, the item at the front of the list. A is still (6, 7, 5).
3. The operation remove(A) returns 6, the item at the front of the list. 6 is no longer in A, so A is now (7, 5).
4. Size(A) returns 2 because there are two items in A.

5. add(3) adds 3 to the back of A. A is now (7, 5, 3).

PARTICIPATION ACTIVITY 7.13.6: Identifying the output of operations on queues.

Match each object to its description. The queues A and B are:

A: (9, 5, 8, 7)
B :(5, 8, 7)

(5, 8, 7, 9)	(9, 5, 8, 7)	5	(5, 8, 7)	3

	The state of A after operation front(A)
	The state of A after operation remove(A)
	The value returned from front(B) operation
	The value returned by size(B)
	The state of B after add(9)

Reset

The merge operation

The input to the merge operation are two sorted lists, A and B. The output is a single list C that contains all the items from A and B in sorted order. The first loop continues to iterate as long as A and B are both non-empty. Each iteration proceeds by selecting the smaller of the two items at the front of A and B, removing that item, and adding the item to the back of C. Loop 1 ends when either A or B becomes empty.

Immediately after Loop 1, only one of A or B has any items left. If A still has items, then Loop 2 removes each item from A and adds the item to the back of C. If B still has items, then Loop 3 removes each item from B and adds the item to the back of C.

In each iteration of the three loops, one item is removed from A or B and added to C. By the end of Loop 3, both A and B are empty. Therefore the total number of iterations in all three loops is exactly n, where n is the total number of items in A and B at the beginning of the merge. Each iteration performs $\Theta(1)$ operations. Therefore the running time of the Merge operations is $\Theta(n)$.

PARTICIPATION ACTIVITY 7.13.7: Simulating the merge operation.

Animation content:

undefined

Animation captions:

1. The input to the Merge operation is A: (3, 5, 9, 15, 17) and B: (7, 11, 12, 20, 23). Start by creating a new empty queue C.
2. In each iteration of Loop 1, the smaller of the items at the front of A and B is removed and added to C. Loop 1 ends when A is empty and B is (20, 21)
3. Since A is empty, there are no iterations in Loop 2.
4. In Loop 3, each item is removed from B and added to C until B is empty.
5. Return(C). The output of the Merge operation is the merged list C.

PARTICIPATION ACTIVITY 7.13.8: Simulating the merge operation.

The input to the merge operation is:
A: (3, 7, 17, 19, 25)
B: (5, 6, 8, 10, 15)

1) How many times does Loop 1 iterate?

Check Show answer

2) How many times does Loop 2 iterate?

Check Show answer

3) How many times does Loop 3 iterate?

Check Show answer

Pseudo-code for mergesort

The pseudo-code for mergesort is given below. The base case occurs when the size of the input list X is 1. If size(X) = n and n \gt 1, then the items in X are split between two sub-lists, L and R. The first \lceil n/2 \rceil items from X are placed in L, and the remaining \lfloor n/2 \rfloor items from X are placed in R. Then L and R are sorted recursively by calls to mergesort. The two sorted sub-lists that

are returned from the recursive calls are sent to the merge operation which returns the final sorted list.

PARTICIPATION ACTIVITY 7.13.9: Simulating the pseudo-code for mergesort.

Animation captions:

1. The input to MergeSort is the list X = (17, 5, 13, 23, 9, 21, 7, 1, 14). Since X has 9 items, the conditions for the base case are not met.
2. Empty list L and R are created. size(X) = n = 9. m = \lceil 9/2 \rceil = 5.
3. Move m = 5 items from X to L. Afterwards, L is (17, 5, 13, 23, 9) and X is (21, 7, 1, 14).
4. Move n - m = 4 items from X to R. Afterwards, R is (21, 7, 1, 14) and X is empty.
5. Make recursive calls to MergeSort on input L and R. The output are sorted sub-lists L: (5, 9, 13, 17, 23) and R: (1, 7, 14, 21).
6. Call Merge on input A and B. Merge returns the merged list C (1, 5, 7, 9, 13, 14, 17, 21, 23). The sorted list C is returned.

PARTICIPATION ACTIVITY 7.13.10: The recurrence relation for the asymptotic complexity of mergesort.

T(n) is defined to be the number of operations performed by MergeSort on an input list of n items.

1) For input list X with n items, how many operations are performed in the first five lines of MergeSort?
   - \Theta(1)
   - \Theta(\log n)
   - \Theta(n)

2) For input list X with n items, how many operations are performed in the two For-loops?
   - \Theta(1)
   - \Theta(n)
   - T(n/2)

3) Which expression best describes the number of operations that are performed during the two recursive calls made by MergeSort?
   - \Theta(n)
   - T(n/2)
   -

- 2 \cdot T(n/2)

4) The running time for the call to Merge is \Theta(n) for an input list with n items.

   Select the recurrence relation that describes the asymptotic complexity of MergeSort.

   - T(n) = T(n/2) + \Theta(1)
   - T(n) = 2 \cdot T(n/2) + \Theta(1)
   - T(n) = T(n/2) + \Theta(n)
   - T(n) = 2 \cdot T(n/2) + \Theta(n)

Additional exercises

Exercise 7.13.1: Recurrence relation for divide-and-conquer FindMin.

About

(a) The pseudo-code for the divide-and-conquer FindMin is given below. Give a recurrence relation that describes the number of operations performed on an input list with n items.

```
FindMin(n, (a_1, a_2, \ldots, a_n) )

If (n = 1), Return(a_1)

m = \lceil n/2 \rceil
List1 := (a_1, a_2, \ldots, a_{m})
List2 := (a_{m+1}, a_{m+2}, \ldots, a_n)
min1 := FindMin(m, List1)
min2 := FindMin(n-m, List2)

If (min1 \lt min2), Return(min1)
Return(min2)
```


Exercise 7.13.2: Recursion tree for mergesort.

About

(a) Give the recursion tree for a call to Mergesort with input list (6, 2, 21, 5, 17, 1, 15). For each call to Mergesort, give the input list to that call, the input and output for the recursive calls made and the final output list for that call.

recursive calls made and the final output list for that call.

Exercise 7.13.3: Adapting mergesort to remove duplicates.

About

(a) Change mergesort so that the algorithm removes duplicate items in the list as well as sorts the remaining items. For example, on input (15, 2, 21, 5, 15, 2, 15), the output should be (2, 5, 15, 21).

Exercise 7.13.4: Simplifying mergesort with a maximum item.

About

(a) Suppose that there is a constant named MaxItem that is the same type as the items to be sorted and is guaranteed to be larger than any of the items in the input list. Use MaxItem to write a simplified merge operation.
Hint: Add MaxItem to the back of the two lists to be sorted.

Exercise 7.13.5: 3-way mergesort.

About

Here is a 3-way version of mergesort:

```
MergeSort(X)

If (??), Return(X)

L1 := createEmptyQueue()
L2 := createEmptyQueue()
L3 := createEmptyQueue()

n := size(X)
m1 := \lceil n/3 \rceil
m2 := \lceil (n-m1)/2 \rceil

For i := 1 to m1
    x := remove(X)
    add(L1, x)
End-for

For i := 1 to m2
    x := remove(X)
    add(L2, x)
End-for

For i := 1 to (n-m1-m2)
    x := remove(X)
    add(L3, x)
End-for

A := MergeSort(L1)
B := MergeSort(L2)
C := MergeSort(L3)

D := 3WayMerge(A, B, C)

Return(D)
```


3wayMerge is an operations that takes as input 3 sorted lists and returns a single sorted lists that contains all the items in the 3 input lists.

(a) Give the sizes of the three sub-lists when the input list has n items, for n = 9, 10, and 11.

(b) What should the condition be for the base case?

(c) Show that if the input list has n items, then the sizes of the three sub-lists are \lfloor n/3 \rfloor or \lceil n/3 \rceil.

Hint: consider three cases: n = 3k, or n = 3k+1, or n = 3k+2, for some integer k.

(d) Give a recurrence relation describing the asymptotic complexity of 3WayMergeSort. You can assume that 3WayMerge runs in \Theta(n) time, where n is the sum of the sizes of the three input lists.

7.14 Divide-and-conquer algorithms: Binary search

This section has been set as optional by your instructor.

Binary search

The goal of a ***search algorithm*** is to find a target item in a list. For example, consider a student trying to find his name in a list of students enrolled in a class. Assume, for simplicity, that no two students have the same name. If the names in the list are in an arbitrary order, then he must start at the beginning of the list and scan through the names until he finds his name or gets to the end of the list. If the class list is sorted alphabetically, then he should be able to find his name more quickly. ***Binary search*** is an efficient algorithm to search for a target item in a sorted list.

The input to binary search is a sorted list of items and a target item x. If x is in the list, then binary search returns the location of x in the list. If x is not in the list, then binary search returns -1.

PARTICIPATION ACTIVITY 7.14.1: Binary search input and output.

Animation captions:

1. On input list (2, 3, 7, 10, 21, 32) and target item x = 18, the target item is not in the list, so the correct output is -1.
2. On input list (2, 3, 7, 10, 21, 32) and target item x = 10, the target item is the 4th item in the list, so the correct output is 4.

PARTICIPATION ACTIVITY 7.14.2: Input and output for binary search.

The input to binary search is the list:

(2, 6, 9, 13, 17, 23, 29, 31, 34, 45)

1) What is the correct output for target x = 18?

Check Show answer

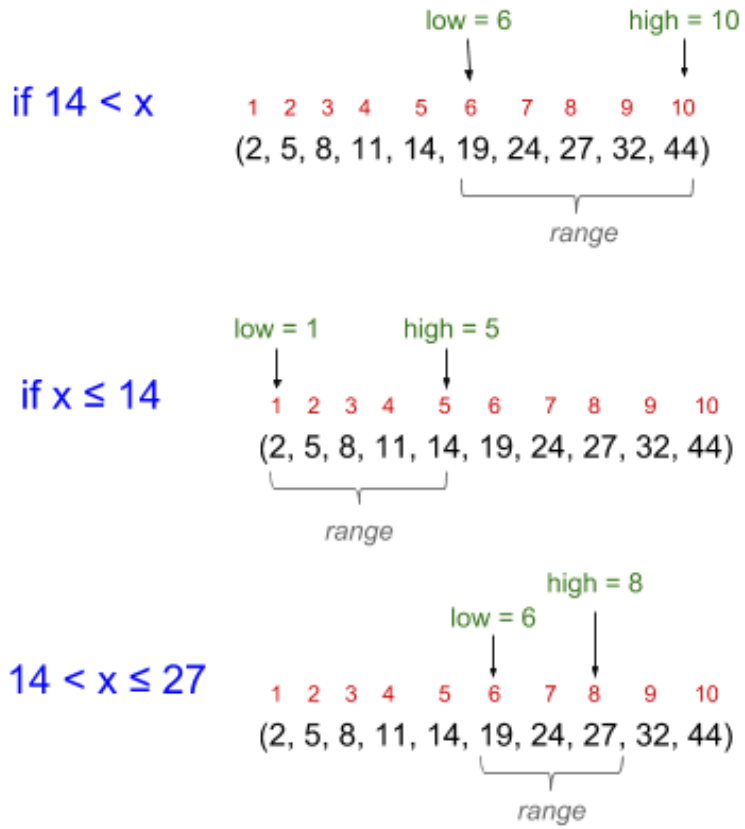
2) What is the correct output for target x = 9?

Check Show answer

Recursive binary search

A recursive version of binary search takes two additional input values, low and high. If n is the number of items in the list, then variables low and high must lie in the range 1 through n and must satisfy the inequality low \le high. The variables low and high denote a range of candidate locations {low, low+1, …, high} so that if x is in the list at all, then x must be reside in one of the locations in the range. For the example L = (2, 4, 7, 10, 21, 32), if 10 \lt x, then x must occur after the 10 in the list. Since the 10 is in location 4, x must occur in location 5 or later in the list and low can be set to 5. If x \le 10, then x must occur in location 4 or earlier in the list, and high can be set to 4. Initially, the algorithm is called with low = 1 and high = n because the range of candidate locations is the entire list.

Figure 7.14.1: Setting variable low and high in binary search.



PARTICIPATION ACTIVITY 7.14.3: Setting variables low and high in binary search.

The input to a recursive binary search algorithm is target x and list L:

(3, 5, 8, 11, 14, 18, 23, 31, 37, 40)

The algorithm sets the variables low and high so that if x occurs in the list, then x is guaranteed to occur somewhere in locations low through high.

1) If 11 \lt x, which assignment is guaranteed to maintain the correct properties of low and high?

- low := 5
- low := 6
- high := 4
- high := 5

2) If x \le 23, which assignment is guaranteed to maintain the correct properties of low and high?

- low := 7
- low := 8
- high := 6

- high := 7

3) If 8 \lt x \le 31, which assignments are guaranteed to maintain the correct properties of low and high?

- low := 4
  high := 8
- low := 3
  high := 7
- low := 5
  high := 9

Pseudo-code

In each recursive call, binary search compares the target item x to the middle item in the range of candidate locations and reduces the size of the range by a half. The base case occurs when the range of candidate locations is down to one (low = high). Then x can be compared to the single item in the range to determine whether x is in the list.

Figure 7.14.2: Pseudo-code for recursive binary search.

```
RecBinarySearch(low, high, A, x)

If (low = high AND a_{low} = x), Return(low)
If (low = high AND a_{low} \neq x), Return(-1)

mid := \lfloor (low + high)/2 \rfloor

If (x \le a_{mid}), then high := mid
If (x \gt a_{mid}), then low := mid + 1

Return( RecBinarySearch(low, high, A, x) )
```

PARTICIPATION ACTIVITY 7.14.4: Recursive binary search simulation.

Animation captions:

1. The input is a sorted list of 7 numbers. low =1 and high = 7. Target x = 14. mid = \lfloor(1 + 7)/2 \rfloor = 4. Compare x to the number in location 4, which is 11.
2. 11 \lt 14, so low becomes mid+1 = 5.

3. mid = \lfloor (5+7)/2 \rfloor = 6. The number in location 5 is 19. Since x \le 19, high is reset to mid = 6.
4. mid = \lfloor (5+6)/2 \rfloor = 5. The number in location 5 is 14. Since x \le 14, high is reset to mid = 5.
5. Since low = high, the conditions for the base case are met. Since x = 14 equals the number in location low, return low. The target number has been found.

PARTICIPATION ACTIVITY 7.14.5: Simulating binary search.

Binary search is called with an input list:

(2, 5, 8, 11, 14, 19, 24)

The target item is x = 6.

For the questions below, enter the values for low and high separated by a comma. For example, for answer low = 3 and high = 7, enter: 3, 7.

```
RecBinarySearch(low, high, A, x)

If (low = high AND a_{low} = x), Return(low)
If (low = high AND a_{low} \neq x), Return(-1)

mid := \lfloor (low + high)/2 \rfloor

If (x \le a_{mid}), then high := mid
If (x \gt a_{mid}), then low := mid + 1

Return( RecBinarySearch(low, high, A, x) )
```

1) What are the values of low and high for the initial call to RecBinarySearch?

Check Show answer

2) What are the values of low and high for the first recursive call to RecBinarySearch?

Check Show answer

3) What are the values of low and high for the second recursive call to

RecBinarySearch?

Check Show answer

4) What are the values of low and high for the last recursive call to RecBinarySearch?

Check Show answer

5) What is the final return value of RecBinarySearch?

Check Show answer

Analyzing recursive binary search

The time complexity of an algorithm is a function of the size of the input. For binary search, the size of the input is the number of candidate locations in the range {low, ..., high}, which is (high - low +1). Since binary search is initially called with low = 1 and high = n, the size of the input is (n - 1 + 1) = n, the number of items in the input list.

Binary search assigns mid := \lfloor (high + low) / 2 \rfloor and breaks the range of locations into two sub-ranges: {low, low+1, ..., mid} and {mid+1, mid+2, ..., high}. The size of the first sub-range is (mid - low + 1). The size of the second sub-range is (high - (mid+1) + 1) = (high - mid). If the variable m denotes the size of the original range (high - low + 1), the fact below indicates that the sizes of the two sub-ranges are m/2 and m/2, in the case that m is even. The sizes of the two sub-ranges are \lceil m/2 \rceil and \lfloor m/2 \rfloor, in the case that m is odd.

Theorem 7.14.1: Sizes of the subranges in binary search.

Suppose that high and low are two positive integers such that low \le high, and define the variable m = (high - low + 1) and mid = \lfloor (high + low) / 2 \rfloor.

- If m is even, then the sizes of the two subranges are both m/2: (mid - low + 1) = (high - mid) = m/2.
- If m is odd, then the size of the first subrange is (mid - low + 1) = \lceil m/2 \rceil, and the size of the second subrange is (high - mid) = \lfloor m/2 \rfloor.

PARTICIPATION 7.14.6: Sizes of the subranges in binary search.

ACTIVITY

Animation captions:

1. When high = 20 and low = 5, the number of locations in the range is 6 which is even. mid = \lfloor (20+15)/2 \rfloor = 17.
2. The two sub-ranges are 15 through 17 and 18 through 20. The size of each sub-grange is 3 = 6/2.
3. When high = 21 and low = 5, the number of locations in the range is 7 which is odd. mid = \lfloor (21+15)/2 \rfloor = 18.
4. The two sub-ranges are 15 through 18 and 19 through 21. The size the first sub-range is 4 = \lceil 7/2 \rceil. The size of the other is 3 = \lfloor 7/2 \rfloor.

PARTICIPATION ACTIVITY 7.14.7: Recurrence relation for binary search.

1) What is the number of recursive calls made in one call of RecBinarySearch for the case that low \lt high?
   - 0
   - 1
   - 2

2) The size of the input to a call to RecBinarySearch is the number of locations in the range {low, ...,high}. If the size of the input to RecBinarySearch is n, then which expression is closest to the size of the input for the recursive call?
   - 1
   - n/2
   - n-1

3) Let T(n) denote the number of atomic operations performed by RecBinarySearch on a list of n items. What is the recurrence relation that describes the asymptotic complexity of RecBinarySearch ?
   - T(n) = T(n/2) + \Theta(n)
   - T(n) = T(n/2) + \Theta(1)
   - T(n) = 2 \cdot T(n/2) + \Theta(n)
   -

$T(n) = 2 \cdot T(n/2) + \Theta(1)$

Additional exercises

Exercise 7.14.1: Simulating binary search.

About

The input list of binary search is:

(2, 5, 8, 10, 13, 19, 21, 32, 37, 52)

For each target value x given below, give the values for variables low and high for each call to BinarySearch. Then give the final value return value.

(a)
x = 13

(b)
x = 1

(c)
x = 35

(d)
x = 52

Exercise 7.14.2: Proving Theorem 18.14.1.

About

(a) Prove the theorem:
Theorem: Suppose that high and low are two positive integers such that low $\le$ high, and define the variable m = (high - low + 1) and mid = $\lfloor$ (high + low) / 2 $\rfloor$.

- If m is even, then the sizes of the two subranges are both m/2: (mid - low + 1) = (high - mid) = m/2.
- If m is odd, then the size of the first subrange is (mid - low + 1) = $\lceil$ m/2 $\rceil$, and the size of the second subrange is (high - mid) = $\lfloor$ m/2 $\rfloor$.

Hint: consider two cases depending on whether the value (high - low + 1) is even or odd. Then, based on whether (high - low + 1) is even or odd, determine whether (high + low) is even or odd. The exact value for mid can then be determined (without the ceiling or floor function) using the following fact:
Fact: If n is even, then $\lceil$ n/2 $\rceil$ = $\lfloor$ n/2 $\rfloor$ = n/2. If n is odd, then $\lceil$ n/2 $\rceil$ = n/2 + (1/2) and $\lfloor$ n/2 $\rfloor$ = n/2 - (1/2).

Exercise 7.14.3: Worst and best case running time for binary search.

About

(a) Consider running binary search on input list:
(2, 5, 8, 11, 15, 18, 23, 29, 31, 35)
In searching for a target item x that is in the list, the running time of the algorithm may differ depending on which item from the list is the target. Give a target item x, selected from the list, that results in the best running time. Given a target item, selected from the list, that results in the worst running time.

(b) For some input lists, the time to search for an item in the list under binary search is the same for all items in the list. Characterize the input lists that have this property.

Exercise 7.14.4: Three-way binary search.

About

The psuedo-code below gives a slightly different version of binary search:

```
RecBinarySearch2(low, high, A, x)

If (??), Return(-1)

mid := \lfloor (low + high)/2 \rfloor

If (x = a_{mid}), then Return( mid )
If (x \lt a_{mid}), then high := mid - 1
If (x \gt a_{mid}), then low := mid + 1

Return( RecBinarySearch2(low, high, A, x) )
```

(a) What is the right condition for the base case?

(b) Consider running RecBinarySearch2 on input list:
(2, 5, 8, 11, 15, 18, 23, 29, 31, 35)
In searching for a target item x that is in the list, the running time of the algorithm may differ depending on which item from the list is the target item. Give a target item x, selected from the list, that results in the best running time. Given a target item, selected from the list, that results in the worst running time.

7.15 Solving linear homogeneous recurrence relations

This section has been set as optional by your instructor.

A sequence that models a dynamical system is often described most naturally by a recurrence relation. Consider the Fibonacci sequence defined by the initial values and recurrence relation given below.

$$\begin{aligned} f_0 &= 0\\ f_1 &= 1\\ f_n &= f_{n-1} + f_{n-2} \quad \text{for} \quad n \ge 2 \end{aligned}$$

Leonardo Fibonacci originally derived the recurrence relation to model the population growth of a colony of rabbits based on the attributes of the rabbits in the colony. Many important properties of a sequence, however, are not evident from a recursive definition. For example, Fibonacci might have wanted to know how large the colony of rabbits would be after 100 months (f_{100}) or how many months it would take for the colony to reach a population of 1000 (the smallest n such that $f_n \ge 1000$). When a recursively defined sequence of numbers $\{s_n\}$ denotes the number of operations performed by a recursive algorithm on an input of size n, the efficiency of the algorithm depends on how s_n grows as a function of n. To answer questions about specific values in the sequence or to understand how the terms grow as a function of the indices, an explicit formula for the sequence is required. That is, we want to express f_n as a function of n, not as a function of the numbers that occur earlier in the sequence.

For example, consider the following recursively defined sequence: $\begin{aligned} g_0 &= 1\\ g_n &= 3g_{n-1} \end{aligned}$ $\{g_n\}$ is an example of a geometric sequence, and the explicit formula for g_n is 3^n. Finding an explicit formula for a recursively defined sequence is called ***solving a recurrence relation***. Actually, any formula of the form $g_n=c\cdot 3^n$ (for any constant c) satisfies the recurrence relation, so there are really an infinite number of solutions to the recurrence relation $g_n = 3\cdot g_{n-1}$. However there is only one solution that also satisfies the initial value $g_0 = 1$. Typically, the expression "solving a recurrence relation" refers to finding the unique sequence that satisfies the recurrence relation as well as a set of initial values.

There are many different kinds of recurrence relations, some of which are very difficult to solve. There are also recurrence relations for which there are no known explicit formula. This material focuses on a particular class of recurrence relations called linear homogeneous recurrence relations. Each number in a sequence defined by a ***linear homogeneous*** recurrence relation is a linear combination of numbers that occur earlier in the sequence.

Definition 7.15.1: Linear homogeneous recurrence relation.

A linear homogeneous recurrence relation of degree k has the following form: $f_n = c_1 f_{n-1} + c_2 f_{n-2} + \cdots + c_k f_{n-k}$, where the c_j's are constants that do not depend on n, and $c_k \neq 0$.

In a ***homogeneous*** recurrence relation, there are no additional terms in the expression for f_n besides the ones that refer to earlier numbers in the sequence. The following table below illustrates some examples. The color in each example highlights why the recurrence relation is either non-linear or non-homogeneous.

Table 7.15.1: Examples illustrating linear and non-linear recurrence relations.

Recurrence relation	Type
\require{color}{b_n = b_{n-1} + {\color{green}(b_{n-2} \cdot b_{n-3})}}	Non-linear
\require{color}{c_n = {\color{green}3n} \cdot b_{n-1}}	Non-linear
\require{color}{d_n = d_{n-1} + {\color{green}(d_{n-2})^2}}	Non-linear
\require{color}{f_n = 3 f_{n-1} - 2 f_{n-2} + f_{n-4} + {\color{red}n^2}}	Linear, degree 4. Non-homogeneous.
\require{color}{g_n = f_{n-1} + f_{n-3} + {\color{red}1}}	Linear, degree 3. Non-homogeneous.
\require{color}{h_n = 2 f_{n-1} - f_{n-2}}	Linear, degree 2. Homogeneous.
\require{color}{s_n = 2 s_{n-1} - \sqrt{3} s_{n-5}}	Linear, degree 5. Homogeneous.

PARTICIPATION ACTIVITY 7.15.1: Identifying linear homogeneous recurrence relations.

Select the type for each of the following recurrence relations.

1) f_n = -f_{n-1} + \sqrt{2}f_{n-5}

- ○ Non-linear.
- ○ Linear, non-homogeneous.
- ○ Linear, homogeneous.

2) f_n = -f_{n-1} + \sqrt{n}f_{n-5}

- ○ Non-linear.
- ○ Linear, non-homogeneous.
- ○

Linear, homogeneous.

3) f_n = -f_{n-1} + 3 f_{n-2} + 1

- Non-linear.
- Linear, non-homogeneous.
- Linear, homogeneous.

To solve a linear recurrence relation, we start with the guess that f_n has the form x^n for some x. While there are mathematical tools to derive the solution to linear homogeneous recurrence relations without resorting to a guess, we know from experience that an exponential function is a good place to start. Plugging in the exponential expression into the recurrence relation, gives rise to an equation called characteristic equation for the recurrence. The ***characteristic equation*** for a linear recurrence relation can be used to solve for x, the base of the exponent in the solution. The following animation illustrates for the Fibonacci recurrence relation:

PARTICIPATION ACTIVITY 7.15.2: A characteristic equation for a linear recurrence relation.

Animation captions:

1. For recurrence relation f_n = f_{n-1} + f_{n-2}, assume f_n = x^n, f_{n-1} = x^{n-1}, and f_{n-2} = x^{n-2}. Then x^n = x^{n-1} + x^{n-2}.
2. Then x^2\cdot x^{n-2} = x\cdot x^{n-2} + 1\cdot x^{n-2}.
3. Divide by x^{n-2} to get x^2 = x + 1, which is the same as x^2 - x - 1 = 0. The final equation is the characteristic equation.

In general the characteristic equation for a degree d linear homogeneous recurrence relation will have the form p(x) = 0, where p(x) is a polynomial of degree d. (The degree of a polynomial is its largest exponent).

PARTICIPATION ACTIVITY 7.15.3: Matching characteristic equations to linear recurrence relations.

Match each characteristic equation to the corresponding recurrence relation.

$f_n = -2 f_{n-1} - 2 f_{n-2}$	$f_n = f_{n-1} + f_{n-5}$	$f_n = 2 f_{n-1} + 2 f_{n-2}$	$f_n = f_{n-4} + f_{n-5}$

	$x^2 - 2x - 2 = 0$
	$x^2 + 2x + 2 = 0$
	$x^5 - x^4 - 1 = 0$

$x^5 - x - 1 = 0$

Reset

Once the characteristic equation has been determined, the next step is to find all values of x that solve the equation. Since the characteristic equation has the form p(x) = 0, for some polynomial p(x), the goal is to find the roots of the polynomial. Some polynomials have complex roots. This material only covers the case where the characteristic equation has real roots. We first consider the case where all the roots are distinct. For example, the characteristic equation for the Fibonacci sequence gives rise to the quadratic equation: x^2 - x - 1 = 0 Solving for x yields two distinct solutons: x = \frac{1 + \sqrt{5}}{2}~~~\mbox{and}~~~x = \frac{1 - \sqrt{5}}{2} Both values for x satisfy the recursive relation. In fact, any linear combination of the two solutions satisfies the recurrence relation $f_n = f_{n-1} + f_{n-2}$. A linear combination of the two solutions has the following form, where the variables s and t can be any real numbers: f_n = s \left(\frac{1 + \sqrt{5}}{2} \right)^n + t \left(\frac{1 - \sqrt{5}}{2} \right)^n Thus, there are an infinite number of solutions to the recurrence relation. However, not all of the solutions satisfy the initial values. The expression denoting the infinite set of solutions to a recurrence relation without initial values is called the ***general solution*** to the recurrence relation. The initial values add constraints that narrow down the set of possible solutions to particular values for s and t. To summarize, if x and y are solutions to the characteristic equations, then $f_n = x^n$ and $f_n = y^n$ both satisfy the recurrence relation. Furthermore any linear combination of x^n and y^n satisfy the recurrence relation as is stated more formally in the theorem below:

Theorem 7.15.1: Solutions to linear homogeneous recurrence relations.

If g_n and h_n satisfy a linear homogeneous recurrence relation then so does $f_n = s \cdot g_n + t \cdot h_n$ for any real numbers s and t.

Proof.

\begin{align*} f_n &= sg_n + th_n\\ &\\ &= s(c_1 g_{n-1} + c_2 g_{n-2} + \cdots + c_d g_{n-d}) + t(c_1 h_{n-1} + c_2 h_{n-2} + \cdots + c_d h_{n-d})\\ &\\ &= c_1(sg_{n-1} + th_{n-1}) + c_2(sg_{n-1} + th_{n-1}) + \cdots + c_d(s g_{n-d} + t h_{n-d})\\ &\\ &=c_1 f_{n-1} + c_2 f_{n-2} + \cdots + c_d f_{n-d} \end{align*} The first equality is the definition of f_n. The second equality uses the fact that g_n and h_n both satisfy the recurrence relation. The third equality just rearranges the terms. The last equality is again the definition of f_n. ■

The solutions to the characteristic equations give a set of solutions to the recurrence relation. The theorem above says that any linear combination of the solutions will also be a solution to the recurrence relation. The final step is to use the initial values to find the exact linear combination that satisfies the recurrence relation and the initial values. If the recurrence relation has degree d, then d initial values are required. The animation below illustrates using the Fibonacci sequence:

PARTICIPATION ACTIVITY 7.15.4: Using initial values to find the unique solution to a recurrence relation.

Animation captions:

1. f_0 = 0. Plug in n = 0 into the general form for f_n: 0 = c_1((1 + \sqrt{5})/2)^0 + c_2((1 - \sqrt{5})/2)^0.
2. Simplifying the expression results in the equation: c_1 + c_2 = 0.
3. f_1 = 1. Plug in n = 1 into the general form for f_n: 1 = c_1((1 + \sqrt{5})/2)^1 + c_2((1 - \sqrt{5})/2)^1.
4. There are two variables (c_1 and c_2) and two linear equations.
5. Solving for c_1 and c_2 results in c_1 = 1/\sqrt{5} and c_2 = 1/\sqrt{5} and f_n = (1/\sqrt{5})((1 + \sqrt{5})/2)^n - (1/\sqrt{5})((1 - \sqrt{5})/2)^n.
6. f_n = (1/\sqrt{5})((1 + \sqrt{5})/2)^n - (1/\sqrt{5})((1 - \sqrt{5})/2)^n is the closed form solution for f_n.

The final closed form solution for the Fibonacci sequence is f_n = \frac{1}{\sqrt{5}}\left(\frac{1 + \sqrt{5}}{2}\right)^{n} - \frac{1}{\sqrt{5}}\left(\frac{1 - \sqrt{5}}{2}\right)^{n} It is clear from the recursive definition of the Fibonacci sequence that every number in the sequence is a positive integer. Each number is the sum of the two previous numbers which are also positive integers. And, yet, the closed form solution contains square roots and negative numbers. Nonetheless, for every positive integer n, the expression above evaluates to a positive integer.

The closed form expression for f_n gives insight to how f_n grows with n. The second root is (1 - \sqrt{5})/2 \approx -.618. The value of (-.618)^{n} goes to zero as n becomes every large. Therefore the expression for f_n is dominated by the first term for large n: f_n \approx \phi^{n}/\sqrt{5}, where \phi = (1 + \sqrt{5})/2.

The constant \phi = (1 + \sqrt{5})/2 has its own name (the ***golden ratio***). The golden ratio describes certain proportions found in natural objects and has been used for thousands of years in man made structures in art and architecture.

The table below shows the steps to solve a degree d linear homogeneous recurrence relation. Each step is illustrated with the following example: f_0 = 7~~~~~~~~~f_1 = 0~~~~~~~~f_2 = 10\\ ~~~~~\\ f_n = 2 f_{n-1} + f_{n-2} - 2 f_{n-3}

Table 7.15.2: Solving linear homogeneous recurrence relations: distinct roots.

General step	Example
Use the recurrence relation to find the characteristic equation which will be p(x) = 0, where p(x) is a degree d polynomial.	\require{color}{\color{green}x^3 - 2x^2 - x +2 = 0}
Find all d solutions to the characteristic equation (i.e., the roots of p(x)). For now, we are assuming that p(x) has d distinct roots $x_1,...,x_d$.	\require{color}{\color{green}x^3 - 2x^2 - x +2 = (x-2)(x+1)(x-1) = 0} \require{color}{\color{green}x_1 = 2,~~~ x_2 = 1, ~~~x_3 = -1}

Every solution f_n = (x_1)^n satisfies the recurrence relation. Therefore, any solution of the formf_n = a_1 (x_1)^n + \cdots + a_d(x_d)^n satisfies the recurrence relation.	\require{color}{\color{green}f_n = a_1 2^n + a_2 1^n + a_3 (-1)^n}
Each initial value gives a value of f_n for a specific value for n. Plug the value for n and f_n into the above expression to get a linear equation with variables $a_1,...,a_d$. There are d linear equations (from the d initial values) and d variables.	\require{color} {\color{green}\begin{align*}n=0: & f_0 = 7 = a_1 + a_2 + a_3 \\ \\ n=1: & f_1 = 0 = 2a_1 + a_2 - a_3 \\ \\ n=2: & f_2 = 10 = 4a_1 + a_2 + a_3 \\ \end{align*}}
Solve for $a_1,...,a_d$.	\require{color}{\color{green} a_1 = 1,~~~a_2 = 2,~~~ a_3 = 4 }
Plug values for $a_1,...,a_d$ back in to the expression for f_n to get the closed form expression for f_n.	\require{color}{\color{green} \begin{align*} f_n & = 1 \cdot 2^n +2 \cdot 1^n + 4 (-1)^n\\ & \\ & = 2^n +2 + 4 (-1)^n \end{align*} }

PARTICIPATION ACTIVITY 7.15.5: Solving linear homogeneous recurrence relations.

1) Consider a linear homogeneous recurrence relation whose characteristic equation is (x - 3)(x + 2) = 0. Which expression characterizes the set of all solutions to the recurrence relation? The variables s and t can be any real numbers.
   - $s \cdot 3^n + t \cdot (-2)^n$
   - $s \cdot 3^n + t \cdot 2^n$
   - $s \cdot (-3)^n + t \cdot 2^n$

2) Suppose that one of the initial values for the previous question is $g_0 = 1$. Which one of the linear equations holds for variables s and t?
   - s + t = 0.
   - s + t =1.
   - 3s - 2t = 1.

3) Suppose that another one of the initial values for the previous question is $g_1 = 3$. Which one of the linear equations holds for variables s and t?

- 3s - 2t = 3.
- 3s - 2t = 1.
- -3s + 2t = 3.

Characteristic equations with multiple roots

Consider a recurrence relation $g_n = 4g_{n-1} - 4g_{n-2}$ whose characteristic equation is x^2 -4x+4 = (x-2)(x-2)=0. 2 is a root of the polynomial x^2 -4x+4 with multiplicity 2. We already know that $g_n = 2^n$ satisfies the recurrence relation. Since 2 is a root of multiplicity 2, $g_n = n2^n$ also satisfies the recurrence relation. The equations below shows the steps required to verify that $g_n = n2^n$ also satisfies $g_n = 4g_{n-1} - 4g_{n-2}$:

- $g_n = n \cdot 2^n$
- $g_{n-1} = (n - 1)2^{n-1}$
- $g_{n-2} = (n - 2)2^{n-2}$
- Plug into recurrence relations: $g_n = 4g_{n-1} - 4g_{n-2}$
- Verify that: $n2^n = 4(n - 1)2^{n-1} - 4(n - 2)2^{n-2}$

Since 2^n and $n2^n$ both satisfy the recurrence relation for g_n, then any linear combination of the two expressions also satisfies the recurrence relation. The final solution will have the form $s2^n + tn2^n$. The initial values can be used as before to solve for s and t. If the initial values are $f_0 = 2$ and $f_1 = 3$, then the resulting linear equations are s = 2 and 2s + 2t = 3:

\require{color} \begin{align*} {\color{red}n = 0:} ~~~~~& f_0 = 2 = s \cdot 2^0 + t \cdot 0 \cdot 2^0 = s\\ &\\ {\color{red}n = 1:} ~~~~~& f_1 = 3 = s \cdot 2^1 + t \cdot 1 \cdot 2^1 = 2s + 2t\\ \end{align*}

The solutions are s = 2 and t = -1/2, so the unique solution that satisfies the recurrence relation and the initial values is:

f_n = 2\cdot 2^n - \frac 1 2 n2^n.

In general if the characteristic equation for a linear recurrence relation is p(x) = 0 and the polynomial p(x) has a root x with multiplicity m, then all of the expressions satisfy the recurrence relation: \require{color} f_n=x^n,~~~f_n={\color{red}n} x^n,~~~f_n={\color{red}n^2} x^n, \ldots, f_n={\color{red}n^{m-1}}x^n. The procedure to solve a linear recurrence relation is to find all the solutions from all of the roots and then take a linear combination of all the solutions. The initial values are used to solve for the constants in the linear combination. The animation below illustrates some examples:

PARTICIPATION ACTIVITY 7.15.6: Linear homogeneous recurrence relations - multiple roots.

Animation captions:

1. Characteristic equation: (x-2)^3(x-3)^2 = 0. x = 2 has multiplicity 3 (2^n, n2^n, n^22^n). x = 3 has multiplicity 2 (3^n, and n3^n). All satisfy the rec relation.
2. g_n = a_1\cdot 2^n + a_2\cdot n2^n + a_3\cdot n^22^n + a_4\cdot 3^n a_5\cdot n3^n is the general form. 5 initial values are required to solve for a_1, a_2, a_3, a_4, a_5.

PARTICIPATION ACTIVITY | 7.15.7: Matching characteristic equations to recurrence relations - multiple roots.

Match each characteristic equation for a recurrence relation to the expression denoting the solutions to the recurrence relation.

$(x + 1)(x - 3)(x - 2)^2$ $(x - 3)^4$ $(x + 1)(x - 3)(x - 4)(x + 2)$

	$a_1(-1)^n + a_2 3^n + a_3 4^n + a_4(-2)^n$
	$a_1 3^n + a_2 n 3^n + a_3 n^2 3^n + a_4 n^3 3^n$
	$a_1(-1)^n + a_2 3^n + a_3 2^n + a_4 n \cdot 2^n$

Reset

Additional exercises

Exercise 7.15.1: Finding characteristic equations.

About

Give the characteristic equation for each of the following recurrence relations:

(a) $a_n = 3a_{n-1} - a_{n-2} + 17a_{n-3}$

(b) $a_n = -a_{n-2} - a_{n-4}$

(c) $a_n = 3a_{n-7}$

Exercise 7.15.2: Finding general solutions for a given characteristic equation.

About

Each equation below is a characteristic equation for a recurrence relation. Express the solution to each recurrence relation as a linear combination of terms. Note that you will

not know the actual coefficients in the linear combination since you are not given the base cases, so you can use a_1, a_2, etc.

(a)
$(x - 1)(x + 2) = 0$

(b)
$(x - 4)(x - 4) = 0$

(c)
$(x - 4)(x - 4)(x + 5) = 0$

(d)
$(x - 4)(x - 4)(x + 5)^4 = 0$

Exercise 7.15.3: Solving linear homogeneous recurrence relations.

About

Solve each of the following recurrence equations with the given initial values.

(a)
$b_n = b_{n-1} + 12b_{n-2}$. Initial values: $b_0 = -2$, $b_1 = 20$.

(b)
$b_n = 3b_{n-1} + 4b_{n-2}$. Initial values: $b_0 = 4$, $b_1 = 1$.

(c)
$b_n = 4b_{n-2}$. Initial values: $b_0 = 2$, $b_1 = 16$.

(d)
$b_n = 4b_{n-1} - 4b_{n-2}$. Initial values: $b_0 = 3$, $b_1 = 10$.

(e)
$b_n = 3b_{n-1} + 4b_{n-2} - 12b_{n-3}$. Initial values: $b_0 = 4$, $b_1 = -5$, $b_2 = 11$.

(f)
$b_n = 5b_{n-1} - 8b_{n-2} + 4b_{n-3}$. Initial values: $b_0 = 6$, $b_1 = 7$, $b_2 = 17$.

7.16 Solving linear non-homogeneous recurrence relations

This section has been set as optional by your instructor.

The associated homogeneous recurrence relation

A ***non-homogeneous linear recurrence relation*** is a linear recurrence relation with additional terms that are either a constant or a function of n. The recurrence relations below are all examples of non-homogeneous linear recurrence relations. $$\begin{aligned} f_n &= 3 f_{n-1} + 10 f_{n-2} + 2 \\ f_n &= 3 f_{n-1} + 10 f_{n-2} + 24n \\ f_n &= 3 f_{n-1} + 10 f_{n-2} + 2^n + 3n \end{aligned}$$ The ***associated homogeneous recurrence relation*** is the recurrence relation with the additional non-homogeneous terms dropped. For example, the associated homogeneous recurrence relation for all of the recurrence relations given above is $f_n = 3 f_{n-1} + 10 f_{n-2}$.

Table 7.16.1: Examples of non-homogeneous linear recurrence relations and their associated homogeneous recurrence relation.

Non-homogeneous recurrence relation	Associated homogeneous recurrence relation
$f_n = 2 f_{n-1} + 12 f_{n-3} + 3 \cdot 7^n + 21$	$f_n = 2 f_{n-1} + 12 f_{n-3}$
$g_n = 3 g_{n-2} + \sqrt{n} + \log n$	$g_n = 3 g_{n-2}$
$h_n = -h_{h-1} - 4 h_{n-4} + n!$	$h_n = -h_{h-1} - 4 h_{n-4}$

PARTICIPATION ACTIVITY 7.16.1: Associated homogeneous recurrence relations.

1) Select the homogeneous recurrence relation associated with

$f_n = 5 f_{n-1} - 3 f_{n-2} + 3 n^2 + 7n - 5$

- $f_n = 5 f_{n-1}$
- $f_n = 5 f_{n-1} - 3 f_{n-2}$
- $f_n = 5 f_{n-1} - 3 f_{n-2} - 5$
- $f_n = 5 f_{n-1} - 3 f_{n-2} + 7n - 5$

Particular and homogeneous solutions

The solution to a non-homogeneous linear recurrence relation is the sum of two parts: a ***homogeneous solution*** plus a ***particular solution***. If the sequence is \{f_n\}, then the homogeneous solution is denoted by f_n^{(h)} and the particular solution is denoted by f_n^{(p)}.

- The homogeneous solution f_n^{(h)} is the general solution to the associated homogeneous recurrence relation. For recurrence relation f_n = 3 f_{n-1} + 10 f_{n-2} + 24n, the homogeneous solution f_n^{(h)} is the general solution to f_n = 3 f_{n-1} + 10 f_{n-2}, whose characteristic equation is x^2 - 3x - 10 = (x-5)(x+2) = 0. Therefore, f_n^{(h)} = a_1 5^n + a_2 (-2)^n. The details of finding a solution to a linear homogeneous recurrence relation are covered elsewhere.
- The method for finding the particular solution f_n^{(p)} is to guess the form of the particular solution and then check the guess. For recurrence f_n = 3 f_{n-1} + 10 f_{n-2} + 24n, the additional term 24n is linear in n. Therefore, a reasonable guess is that the particular solution is also linear. The form of a linear solution is (an + b) for some constants a and b. The form (an + b) is valid if it is possible to solve for constants a and b so that f_n = (an + b) satisfies the recurrence relation for every value of n.

PARTICIPATION ACTIVITY 7.16.2: Checking a particular solution.

Animation captions:

1. For recurrence f_n = 3 f_{n-1} + 10 f_{n-2} + 24n, guess particular solution f^{(p)}_n = an + b.
2. Plug the particular solution into the recurrence to solve for a and b. The equation must be true for all n if f_n^{(p)} = an + b is a valid particular solution.
3. Collect linear terms in n to solve for a.
4. Collect the constant terms (which do not have a factor of n). Plug in the value for a in order to solve for b.
5. Plug values a = -2 and b = -23/6 back into the particular solution f_n^{(p)} = an + b = -2n - (23/6).

PARTICIPATION ACTIVITY 7.16.3: Finding the constants in the form of a particular solution.

Consider the recurrence relation

f_n = 5 f_{n-1} - 6 f_{n-2} + 7^n.

The non-homogeneous term is 7^n so the guess for the form of the particular solution will be c 7^n, for some constant c.

1) Select the equation that results from plugging in

   f_n = c 7^n

f_{n-1} = c 7^{n-1}
f_{n-2} = c 7^{n-2}

into the recurrence relation.

- c 7^n = 5c 7^{n-1} - 6c 7^{n-2} + c 7^n
- c 7^n = 5c 7^{n} - 6c 7^{n} + c 7^n
- c 7^n = 5 \cdot 7^{n-1} - 6 \cdot 7^{n-2} + 7^n
- c 7^n = 5c 7^{n-1} - 6c 7^{n-2} + 7^n

2) Select the equation that results from taking the answer to the previous question and dividing both side by 7^{n-2}.

- 49c = 35c - 6c + 1
- 49c = 7c - 6c + 49
- 49c = 35c - 6c + 49
- 49c = 35c - 6c + 49c

3) The answer to the previous question can be simplified as 20c = 49. What is the particular solution for the recurrence relation?

- f_n = \frac{49}{20} \cdot 7^n
- f_n = 7^{49n/20}
- f_n = \frac{20}{49} \cdot 7^n

Using the initial conditions to derive the final solution

The recurrence relation f_n = 3 f_{n-1} + 10 f_{n-2} + 24n has homogeneous solution f_n^{(h)} = a_1 5^n + a_2 (-2)^n and particular solution f_n^{(p)} = -2n - \frac{23}{6}. The general solution to the recurrence relation is: f_n = f_n^{(h)} + f_n^{(p)} = a_1 5^n + a_2 (-2)^n -2n - \frac{23}{6} The final step is to use the initial conditions to solve for the constants a_1 and a_2. Suppose that the initial conditions are: f_0 = \frac{7}{6}~~~~~~f_1 = \frac{-11}{6} Each of the initial values results in a linear equation. The first equation is determined by plugging in n = 0 into the general solution for f_n and setting the resulting expression equal to f_0 = 7/6. The second equation is determined by plugging in n = 1 into the general solution for f_n and setting the resulting expression equal to f_1 = -11/6. The two equations are linear in a_1 and a_2 and can be used to solve for a_1 and a_2.

PARTICIPATION ACTIVITY 7.16.4: Solving for the constants in a general solution.

Animation content:

undefined

Animation captions:

1. The general solution is f_n = a_1 5^n + a_2 (-2)^n - 2n - \frac{23}{6}. The initial conditions are f_0 = 7/6 and f_1 = -11/6.
2. Plug in n = 0 to the general solution. The resulting expression a_1 5^0 + a_2 (-2)^0 - 2 \cdot 0 - \frac{23}{6} is equal to the initial value of f_0 = \frac{7}{6}.
3. Simplifying the equation for n = 1 results in the equation -\frac{11}{6} = a_1 + a_2 - \frac{23}{6}.
4. Plug in n = 1 into the general solution. The resulting expression 5 a_1 - 2 a_2 - 2 - \frac{23}{6} is equal to the initial value of f_1 = -\frac{11}{6}.
5. Solving the two linear equations results in a_1 = 2 and a_2 = 3. Plugging the values for a_1 and a_2 back into the general solution results in the final solution: f_n = 2 \cdot 5^n + 3 (-2)^n - 2n - \frac{23}{6}.

PARTICIPATION ACTIVITY 7.16.5: Linear equations to find constants in the general solution.

The general solution to

f_n = 5 f_{n-1} - 6 f_{n-2} + 7^n

is

f_n = a_1 2^n + a_2 3^n + \frac{49}{20} \cdot 7^n

1) What is the linear equation that results from the initial value f_0 = 11?
   - 11 = a_1 + a_2 + \frac{49}{20}
   - 11 = 2 a_1 + 3 a_2 + \frac{49}{20}
   - 11 = a_1 + a_2 + \frac{49}{20} \cdot 7

2) What is the linear equation that results from the initial value f_1 = 13?
   - 13 = a_1 + a_2 + \frac{49}{20}
   - 13 = 2 a_1 + 3 a_2 + \frac{49}{20}
   - 13 = a_1 + a_2 + \frac{49}{20} \cdot 7
   - 13 = 2 a_1 + 3 a_2 + \frac{49}

{20} \cdot 7

Summary of all the steps

The table below summarizes the steps for solving a linear non-homogeneous recurrence relation. The column on the right shows the result of each step on the example f_n = 3 f_{n-1} + 10 f_{n-2} + 24n with initial values f_0 = 7/6 and f_1 = -11/6.

Table 7.16.2: Solving linear non-homogeneous recurrence relations.

General step	Example: f_n = 3 f_{n-1} + 10 f_{n-2} + 24n
Find the homogeneous part of the solution which is the general solution to the associated homogeneous recurrence relation.	The homogeneous part is \require{color} {\color{green}f^{(h)}_n = a_1 5^n + a_2 (-2)^n} which is the general solution to \require{color} {\color{green}f_n = 3 f_{n-1} + 10 f_{n-2}}
Guess the correct form for the particular solution.	Guess \require{color} {\color{green}f^{(p)}_n = an + b} for some constants a and b.
Verify the guess by solving for constants so that the guess satisfies the recurrence relation for all n.	Find constants a and b so that \require{color}{\color{green}an + b = 3(a(n-1)+b) + 10(a(n-2) + b) + 24n} for every n. \require{color} {\color{green}a = -2~~~~~b = \frac{-23} 6}
Add the homogeneous and particular solutions to get the general solution.	\require{color} {\color{green}\begin{align*}f_n & = f_n^{(h)} + f_n^{(p)}\\ & \\ & = a_1 5^n + a_2 (-2)^n - 2n - \frac{23}{6} \end{align*}}
For a degree d linear recurrence relation, there must be d initial values to specify the sequence. Each initial value gives a value of f_n for a specific value for n. Plug the values for n and f_n into the general solution to get a linear equation with variables a_1, \ldots, a_d. There are d linear equations (from the d initial values) and d variables.	\require{color} {\color{green}\begin{align*}n=0: & f_0 = \frac{7}{6} = a_1 + a_2 - \frac{23}{6} \\ \\ n=1: & f_1 = \frac{-11}{6} = 5 a_1 - 2a_2 - \frac{35}{6}\\ \end{align*}}
Solve for a_1, \ldots, a_d.	\require{color}{\color{green} a_1

	= 2,~~~a_2 = 3 }
Plug values for a_1, \ldots, a_d back in to the general solution for f_n to get the final closed form expression for f_n.	\require{color}{\color{green} f_n = 2 \cdot 5^n + 3 (-2)^n - 2n - \frac{23}{6}}

PARTICIPATION ACTIVITY 7.16.6: Solving linear non-homogeneous recurrence relations.

The homogeneous solution to

g_n = 6 g_{n-1} - 9 g_{n-2} + 16

is

g^{(h)}_n = a_1 3^n + a_2 n 3^n

1) Since the non-homogeneous term in the recurrence relation is a constant (+16), the guess for the form of the particular solution will be that g_n^{(p)} = c, for some constant c. Select the equation to use to solve for c.
   - c = 6c - 9c + 16
   - c = 6c - 9c + 16c
   - c = 6(c-1) - 9(c-2) + 16

2) What is the general solution to the recurrence relation g_n = 6 g_{n-1} - 9 g_{n-2} + 16?
   - g_n = a_1 3^n + a_2 n 3^n + 16
   - g_n = a_1 3^n + a_2 n 3^n + 4
   - g_n = a_1 3^n + a_2 n 3^n
   - g_n = 4.

3) What is the linear equation that results from the initial value g_0 = 5?
   - 5 = a_1 + 4
   - 5 = a_1 + a_2 + 4
   - 5 = 3 a_1 + 3 a_2 + 4

4) What is the linear equation that results from the initial value g_1 = 13?

- 13 = a_1 + a_2 + 4
- 13 = 3 a_1 + 9 a_2 + 4
- 13 = 3 a_1 + 3 a_2 + 4

Determining the form of the particular solution

There is no rule for determining the form of the particular solution for a linear non-homogeneous recurrence relation that works in every situation. However, there is a rule that works when the non-homogeneous terms have a particular common form. Suppose the recurrence relation is f_n = c_1 f_{n-1} + c_2 f_{n-2} + \cdots c_{d} f_d + F(n) The function F(n) includes all the non-homogeneous terms. The theorem below provides a way to determine the form of the specific solution when F(n) is a polynomial times an exponential: p(n) \cdot s^n for some constant s. The polynomial p(n) can be a degree 0 polynomial, which is just a constant. Also, the value of s can be 1, in which case F(n) is a polynomial.

Theorem 7.16.1: The form of particular solutions to certain linear non-homogeneous recurrence relations.

Suppose the sequence \{f_n\} is described by the linear non-homogeneous recurrence relation f_n = c_1 f_{n-1} + c_2 f_{n-2} + \cdots c_{d} f_d + F(n) and suppose that the function F(n) has the form F(n) = p(n) s^n, where p(n) is a polynomial of degree t and s is a constant.

- If s is not a root of the characteristic equation for the associated homogeneous recurrence relation, then there is a particular solution of the form f_n = (d_t n^t + d_{t-1} n^{t-1} + \cdots + d_1 n + d_0) s^n
- If s is a root of the characteristic equation for the associated homogeneous recurrence relation of multiplicity m, then there is a particular solution of the form f_n = n^m(d_t n^t + d_{t-1} n^{t-1} + \cdots + d_1 n + d_0) s^n

PARTICIPATION ACTIVITY 7.16.7: Determining the form of the particular solution.

Animation captions:

1. For recurrence f_n = 5 f_{n-1} - 6 f_{n-2} + 3n^2 + 6, the associated homogeneous recurrence has characteristic equation x^2 - 6x + 6 = (x-2)(x-3) = 0.
2. F(n) is a degree 2 polynomial times 5^n, and 5 is not a root of the characteristic polynomial.
3. The form of the particular solution is a degree 2 polynomial (an^2 + bn + c)5^n. The recurrence relation can be used to solve for constants a, b, and c.
4. For recurrence f_n = 8 f_{n-1} - 21 f_{n-2} + 18 f_{n-3}+ 3^n, the associated homogeneous recurrence has characteristic equation x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2 = 0.

5. F(n) is a degree 0 polynomial (which is a constant) times 3^n. 3 is a root of the characteristic polynomial with multiplicity 2.
6. The form of the particular solution is f_n^{(p)} = c n^2 3^n. The recurrence relation can be used to solve for the constant c.

PARTICIPATION ACTIVITY 7.16.8: Determining the form of the particular solution.

Animation captions:

1. For recurrence f_n = -6 f_{n-1} + 7 f_{n-2} + 3n^2 + 6, the associated homogeneous recurrence has characteristic equation x^2 + 6x -7 = (x+7)(x-1) = 0.
2. F(n) is a degree 2 polynomial times 1^n, and 1 is a root of the characteristic polynomial with multiplicity 1.
3. The form of the particular solution is n^1 times a degree 2 polynomial an^2 + bn + c. The recurrence relation can be used to solve for constants a, b, and c.
4. For recurrence f_n = 6 f_{n-1} - 8 f_{n-2} + 6, the associated homogeneous recurrence has characteristic equation x^2 - 6x + 8 = (x-2)(x-4) = 0.
5. F(n) is a degree 0 polynomial (which is a constant) times 1^n, and 1 is not a root of the characteristic polynomial.
6. The form of the particular solution is f_n^{(p)} = c. The recurrence relation can be used to solve for the constant c.

PARTICIPATION ACTIVITY 7.16.9: The form for the particular solution.

Match the form for a particular solution to its corresponding recurrence relation.

f_n^{(p)} = n^2(an^2 + bn + c) | f_n^{(p)} = cn(-3)^n | f_n^{(p)} = (an + b)3^n | f_n^{(p)} = (an + b) | f_n^{(p)} = cn

	f_n = - f_{n-1} + 5 f_{n-2} - 3f_{n-3} + n^2 Characteristic equation: (x - 1)^2(x + 3) = 0
	f_n = -2 f_{n-1} + 3f_{n-2} + 7 Characteristic equation: (x - 1)(x + 3) = 0
	f_n = -2 f_{n-1} + 3f_{n-2} + 5n \cdot

	3^n Characteristic equation: (x - 1)(x + 3) = 0
	f_n = -2 f_{n-1} + 3f_{n-2} + (-3)^n Characteristic equation: (x - 1)(x + 3) = 0
	f_n = 7 f_{n-1} - 12f_{n-2} + 4n Characteristic equation: (x - 3)(x - 4) = 0

Reset

Additional exercises

Exercise 7.16.1: The form of the particular solution.

About

Give the form for the particular solution for each recurrence relation. You do not have to solve for the coefficients. For example, your answer can have the form 3^n(an^2 + bn + c).

(a)
f_n = 8 f_{n-1} - 5 f_{n-2} - 50 f_{n-3} + 7^n n^2

(b)
f_n = 8 f_{n-1} - 5 f_{n-2} - 50 f_{n-3} + 5^n n

(c)
f_n = 8 f_{n-1} - 5 f_{n-2} - 50 f_{n-3} + 2^n

(d)
f_n = 8 f_{n-1} - 5 f_{n-2} - 50 f_{n-3} + 3n^2 - 4

(e)
f_n = 8 f_{n-1} - 5 f_{n-2} - 50 f_{n-3} + 17n

(f)
f_n = -6 f_{n-1} + 7 f_{n-2} + 3^n

(g)
f_n = -6 f_{n-1} + 7 f_{n-2} + 5n^2

(h)

f_n = -6 f_{n-1} + 7 f_{n-2} + 2

Exercise 7.16.2: Solving linear non-homogeneous recurrence relations.

About

Solve the recurrence relation.

(a) f_n = 8 f_{n-1} - 15 f_{n-2} + 2
f_0 = 21/4
f_1 = 85/4

(b) f_n = - f_{n-1} + 6 f_{n-2} + 4n
f_0 = \dfrac{1}{4}
f_1 = -\dfrac{11}{4}

(c) f_n = 8 f_{n-1} - 15 f_{n-2} + 3n^2 - (1/8)
f_0 = 25
f_1 = 359/16

(d) f_n = -2 f_{n-1} + 3 f_{n-2} + 5n - 3
f_0 = 3
f_1 = -57/16

(e) f_n = - f_{n-1} + 5 f_{n-2} - 3 f_{n-3} + 8
f_0 = 5
f_1 = -7
f_2 = 15

(f) f_n = 5 f_{n-1} + 14 f_{n-2} + 3^n
f_0 = 71/20
f_1 = -7/20

(g) f_n = f_{n-1} + 6 f_{n-2} + 3 \cdot (-2)^n
f_0 = 10
f_1 = -12/5

7.17 Divide-and-conquer recurrence relations

This section has been set as optional by your instructor.

Deriving solutions to divide-and-conquer recurrence relations

Many recurrence relations that describe the time complexity of divide-and-conquer algorithms have the form $T(n) = a\ T(n/b) + \Theta(n^d)$ On a call to the algorithm with input of size n, $\Theta(n^d)$ operations are performed outside the recursive calls. In addition, there are a recursive calls made on inputs of size n/b. For example, the recurrence relation for mergesort is $T(n) = 2\ T(n/2) + \Theta(n)$. Mergesort makes 2 recursive calls to sub-lists of size n/2 and then performs $\Theta(n)$ work in merging the two sorted sublists.

The animation below illustrates the total amount work done by a divide-and-conquer algorithm whose recurrence relation is $T(n) = 3\ T(n/2) + n^d$ for some constant d. The initial value for T is $T(1) = 1$. The analysis ignores the complication of having to round n/2 to $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$ and simply uses the term n/2 to represent either possibility. The simplifications does not affect the asymptotic growth of the explicit formula for T(n).

The analysis proceeds by expanding the terms using the recurrence relation: Every expression T(m) is replaced by three $T(m/2)$ terms and one m^d term because $T(m) = 3\ T(m/2) + m^d$. The expansion continues until the initial value T(1) is reached.

PARTICIPATION ACTIVITY 7.17.1: Expanding the terms in a divide-and-conquer recurrence relation: Number of levels.

Animation captions:

1. The value of T(n) is the sum of four terms: $n^d + T(n/2) + T(n/2) + T(n/2)$. At level 0, replace T(n) by the sum of the four terms.
2. At level 1, the fact that $T(n/2) = 3T(n/4) + (n/2)^d$, can be used to replace each of the T(n/2) terms in the expression for T(n) with four terms: $(n/2)^d + T(n/4) + T(n/4) + T(n/4)$.
3. Keep using the recurrence relation to replace each $T(\cdot)$ term until the last level, when the input to T is $n/2^L = 1$. Then apply the initial condition $T(1) = 1$.
4. The number of levels is L+1, where $n/2^L = 1$. Solving for L gives that $L = \log_2 n$.

PARTICIPATION ACTIVITY 7.17.2: Expanding the terms in a divide-and-conquer recurrence relation: Evaluating the sum.

Animation captions:

1. The value of T(n) is the sum of all the terms in the boxes.
2. There is $1 = 3^0$ term at level 0. The value of the term is $n^d = (n/2^0)^d$. Add $3^0 \cdot (n/2^0)^d$ to the sum for T(n).
3. There are $3 = 3^1$ terms at level 1. The value of each term is $(n/2)^d = (n/2^1)^d$. Add $3^1 \cdot (n/2^1)^d$ to the sum for T(n).
4. There are $9 = 3^2$ terms at level 2. The value of each term is $(n/4)^d = (n/2^2)^d$. Add $3^2 \cdot (n/2^2)^d$ to the sum for T(n).

5. Continue until the last level L, where n/2^L = 1. There are 3^L terms at level L. The value of each term is 1 = (n/2^L)^d. Therefore, 3^L \cdot (n/2^L)^d is the last term in the sum for T(n).

PARTICIPATION ACTIVITY 7.17.3: The sum of terms at level j in the expansion.

The questions below refer to the expansion of the recurrence relation T(n) = 3 T(n/2) + n^d shown in the animations above.

1) If the number of terms at level j is m, then how many terms are at level j+1?
   - m/2
   - m/3
   - 2m
   - 3m

2) If there is 1 term at level 0, then how many terms are at level j?
   - n/2^j
   - 2^j
   - 3^j

3) If the value of each term at level j is s^d, then what is the value of each term at level j+1?
   - (s/2)^d
   - (s/3)^d
   - (2s)^d
   - (3s)^d

4) If the term at level 0 is n^d, then what is the value of each term at level j?
   - (n/2^j)^d
   - (n/3^j)^d
   - (3^j)^d

5) What is the sum of the values of all the terms at level j?
   - 2^j (n/3^j)^d
   - 3^j (n/2^j)^d
   - n^d

Expressing the formula as a geometric sum

The function T described by recurrence relation T(n) = 3 T(n/2) + n^d and initial value T(1) = 1 has an explicit formula that is a sum: T(n) = \sum_{j = 0}^L 3^j \left(\frac{n}{2^j} \right)^d~~~~~~~\mbox{where}~~L = \log_2 n The sum can be simplified as T(n) = \sum_{j = 0}^L 3^j \left(\frac{n}{2^j} \right)^d = \sum_{j = 0}^L 3^j \frac{n^d}{2^{jd}} = \sum_{j = 0}^L n^d \left(\frac{3}{2^d} \right)^j = n^d \cdot \sum_{j = 0}^L \left(\frac{3}{2^d} \right)^j The explicit formula for T(n) can be generalized for a recurrence relation of the form T(n) = a T(n/b) + n^d which has an explicit formula T(n) = n^d \cdot \sum_{j = 0}^L \left(\frac{a}{b^d} \right)^j ~~~~~~~\mbox{where}~~L = \log_b n The expression for T(n) is n^d times the sum of terms in a geometric sequence of the form 1 + r + r^2 + \cdots + r^L, where the ratio r = a/b^d. The formula for the sum of a geometric sequence is 1 + r + r^2 + \cdots + r^L = \sum_{j=0}^L r^j = \frac{r^{L+1} - 1}{r-1} The asymptotic growth rate of T(n) depends on whether the ratio r is less than, greater than, or equal to 1.

PARTICIPATION ACTIVITY 7.17.4: Asymptotic growth of the sum of a geometric sequence.

Animation captions:

1. If r \gt 1, then the dominant term in the sum 1 + r + r^2 + \cdots + r^L is r^L. The value of the sum is (r^{L+1} - 1)/(r - r), which is \Theta(r^L).
2. When r = 1, all the terms in the sum are equal to 1. There are L+1 terms, so the value of the sum is L+1, which is \Theta(L).
3. If r \lt 1, then the dominant term in the sum 1 + r + r^2 + \cdots + r^L is 1. The value of the sum is (r^{L+1} - 1)/(r - r), which is \Theta(1).

PARTICIPATION ACTIVITY 7.17.5: Determining the growth rate of the sum of a geometric sequence.

Match each sum to the correct asymptotic growth rate.

\sum_{j=0}^L \left(\frac 1 3 \right)^j

\sum_{j=0}^L 1^j

\sum_{j=0}^L 3^j

	\Theta(3^L)
	\Theta(L)
	\Theta(1)

Reset

PARTICIPATION

ACTIVITY 7.17.6: The asymptotic growth of divide-and-conquer recurrence relations.

1) Select the correct asymptotic growth rate for the expression $n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d} \right)^j$ for a = 4, b = 2, and d = 2.

   - $\Theta(\log n)$
   - $\Theta(n^2)$
   - $\Theta(n^3)$
   - $\Theta(n^2 \log n)$

2) Select the correct asymptotic growth rate for the expression $n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d} \right)^j$ for a = 4, b = 2, and d = 3.

   - $\Theta(n^3)$
   - $\Theta(n^3 \log n)$
   - $\Theta(n^4)$
   - $\Theta(n^4 \log n)$

3) Select the correct asymptotic growth rate for the expression $n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d} \right)^j$ for a = 3, b = 2, and d = 1.

   - $\Theta \left(n \cdot \left(\frac 3 2 \right)^{\log_2 n} \right)$
   - $\Theta\left(n \cdot \left(\frac 3 2 \right)^{n} \right)$
   - $\Theta(n)$
   - $\Theta(n \log n)$

4) Select the expression that is equivalent to $n \left(\frac 3 2\right)^{\log_2 n}$ You will need the following two laws of logarithms: For $a \gt 0$ and $b \gt 1$, $a^{\log_b n} = n^{\log_b a}$ $\log_b b = 1$

   - $n^{\log_2 3}$
   - $n^{\log_2 3 - 1}$
   -

n^{\log_2 3 + 1}

The Master Theorem

The final closed form solution to a divide-and-conquer recurrence relation of the form T(n) = a T(n/b) + \Theta(n^d) is summarized in the Master Theorem which can be used to find the asymptotic time complexity for most divide-and-conquer algorithms. The initial values for the recurrence are always assumed to be \Theta(1).

Although the analysis above used the simpler recurrence T(n) = a T(n/b) + n^d instead of T(n) = a T(n/b) + \Theta(n^d) and the simpler initial value T(1) = 1, instead of T(1) = O(1), the asymptotic growth rate of T is the same.

Theorem 7.17.1: The Master Theorem.

If T(n) = a T(n/b) + \Theta(n^d) for constants a \gt 0, b \gt 1 and d \ge 0, then

- If (a/b^d) \lt 1, then T(n) = \Theta(n^d)
- If (a/b^d) = 1, then T(n) = \Theta(n^d \log n)
- If (a/b^d) \gt 1, then T(n) = \Theta(n^{\log_b a})

PARTICIPATION ACTIVITY 7.17.7: Applying the Master Theorem: An example for each case.

Animation captions:

1. The recurrence relation for binary search is T(n) = T(n/2) + \Theta(1). a = 1, b = 2, and d = 0. Since (a/b^d) = 1, T(n) = \Theta(n^d \log n) = \Theta(\log n).
2. For recurrence relation, T(n) = 4T(n/2) + \Theta(n). a = 4, b = 2, and d = 1. Since (a/b^d) \gt 1, T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2).
3. For recurrence relation, T(n) = 4T(n/3) + \Theta(n^2). a = 4, b = 3, and d = 2. Since (a/b^d) \lt 1, T(n) = \Theta(n^d) = \Theta(n^2).

PARTICIPATION ACTIVITY 7.17.8: Applying the Master Theorem to divide-and-conquer recurrence relations.

Match each recurrence relation with its corresponding growth rate.

\Theta(n \log n) \Theta(n^3) \Theta(n) \Theta(n^{\log_2 7}) \Theta(n^{\log_3 2})

	T(n) = 3 T(n/3) + \Theta(n)
	T(n) = T(n/3) + \Theta(n)
	T(n) = 7 T(n/2) + \Theta(n^2)
	T(n) = 7 T(n/2) + \Theta(n^3)
	T(n) = 2 T(n/3) + \Theta(1)

Reset

Additional exercises

Exercise 7.17.1: Exact solutions for divide-and-conquer recurrence relations.

About

Expand the terms of each recurrence relation in order to obtain an exact solution for T(n). Your solution should include all the constants in the expression for T(n), and not just the asymptotic growth of the function T(n). You can assume that the value of n, the input to the function T, is a power of 3. That is, n = 3^k for some integer k.

(a) T(n) = 3 T(n/3) + 5n

T(1) = 5

(b) T(n) = 3 T(n/3) + 5n^2

T(1) = 5

(c) T(n) = 9 T(n/3) + 5n

T(1) = 5

Exercise 7.17.2: Applying the Master Theorem.

About

Give the asymptotic growth of T(n) using \Theta notation.

(a)

T(n) = 4 T(n/3) + \Theta(n)

(b)

$T(n) = 4 T(n/4) + \Theta(\sqrt{n})$

(c)

$T(n) = 4 T(n/2) + \Theta(n^2)$

(d)

$T(n) = 4 T(n/2) + \Theta(n^3)$

(e)

$T(n) = 2 T(n/3) + \Theta(n)$

(f)

$T(n) = 2 T(n/3) + \Theta(1)$

(g)

$T(n) = 7 T(n/4) + \Theta(n^2)$

(h)

$T(n) = 7 T(n/4) + \Theta(n)$

(i)

$T(n) = 2 T(n/4) + \Theta(\sqrt{n})$

(j)

$T(n) = 3 T(n/3) + \Theta(1)$

©zyBooks 11/09/20 01:05 695959
Gowtham Rajeshshekaran
NYUCSBRTalSummer2020